

USENIX

conference

proceedings

General Track: 2005 USENIX Annual Technical Conference

Anaheim, CA, USA, April 10-15, 2005

General Track

2005 USENIX Annual Technical Conference

Anaheim, CA, USA

April 10-15, 2005

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: <http://www.usenix.org>

The price is \$40 for members and \$50 for nonmembers.
Outside the U.S.A. and Canada, please add
\$20 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

2004	Boston, MA	1991	Winter Dallas
2003	San Antonio, TX	1990	Summer Anaheim
2002	Monterey, CA	1990	Winter Washington, D.C.
2001	Boston, MA	1989	Summer Baltimore
2000	San Diego, CA	1989	Winter San Diego
1999	Monterey CA	1988	Summer San Francisco
1998	New Orleans	1988	Winter Dallas
1997	Anaheim	1987	Summer Phoenix
1996	San Diego	1987	Winter Washington, D.C.
1995	New Orleans	1986	Summer Atlanta
1994	Summer Boston	1986	Winter Denver
1994	Winter San Francisco	1985	Summer Portland
1993	Summer Cincinnati	1985	Winter Dallas
1993	Winter San Diego	1984	Summer Salt Lake City
1992	Summer San Antonio	1984	Winter Washington, D.C.
1992	Winter San Francisco	1983	Summer Toronto
1991	Summer Nashville	1983	Winter San Diego

© 2005 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-27-7

USENIX Association

**Proceedings of the
General Track**

2005 USENIX Annual Technical Conference

**April 10–15, 2005
Anaheim, CA, USA**

Conference Organizers

Program Chair

Vivek Pai, *Princeton University*

Program Committee

Atul Adya, *Microsoft Research*

David Andersen, *Massachusetts Institute of Technology*

Andrea Arpaci-Dusseau, *University of Wisconsin, Madison*

Ludmila Cherkasova, *Hewlett-Packard Labs*

Tzi-cker Chiueh, *Stony Brook University*

Carla Ellis, *Duke University*

Dawson Engler, *Stanford University*

Steven Gribble, *University of Washington*

Erich Nahum, *IBM Research*

Jason Nieh, *Columbia University*

Timothy Roscoe, *Intel Research*

Mema Roussopoulos, *Harvard University*

Michael Stumm, *University of Toronto*

Joe Sventek, *University of Glasgow*

Mustafa Uysal, *Hewlett-Packard Labs*

Hui Zhang, *Carnegie Mellon University*

Yuanyuan Zhou, *University of Illinois at Urbana-Champaign*

Invited Talks Committee

Ethan Miller, *University of California, Santa Cruz*

Ellie Young, *USENIX*

Erez Zadok, *Stony Brook University*

Guru Is In Coordinator

Rob Kolstad, *USENIX*

Poster Session Chair

Atul Adya, *Microsoft Research*

Work-in-Progress Session Chair

David Andersen, *Massachusetts Institute of Technology*

The USENIX Association Staff

External Reviewers

Martin Arlitt

Sujata Banerjee

Ranjita Bhagwan

Prashanth Bungale

Jeff Collard

Fred Douglass

Kave Eshghi

Geoff Goodell

Shlomo HersHKop

Mahesh Kallahalla

Angelos Keromytis

Sanjeev Kumar

Dan Magenheimer

Kostas Magoutis

Shailabh Nagar

Raj Rajagopalan

John Reumann

Marcel Rosu

Yasushi Saito

Anees Shaikh

Alex Sherman

Yoshio Turner

Ke Wang

Limin Wang

Kevin Wilkinson

2005 USENIX Annual Technical Conference
General Track
April 10–15, 2005
Anaheim, CA, USA

Index of Authors	vii
Message from the Program Chair	ix

Wednesday, April 13, 2005

Debugging

Debugging Operating Systems with Time-Traveling Virtual Machines	1
<i>Samuel T. King, George W. Dunlap, and Peter M. Chen, University of Michigan</i>	
Using Valgrind to Detect Undefined Value Errors with Bit-Precision	17
<i>Julian Seward, OpenWorks LLP; Nicholas Nethercote, University of Texas at Austin</i>	
Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution	31
<i>Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin, Duke University</i>	

Planning & Management

Surviving Internet Catastrophes	45
<i>Flavio Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo, and Geoffrey M. Voelker, University of California, San Diego</i>	
Making Scheduling “Cool”: Temperature-Aware Workload Placement in Data Centers	61
<i>Justin Moore and Jeff Chase, Duke University; Parthasarathy Ranganathan and Ratnesh Sharma, Hewlett-Packard Labs</i>	
CHAMELEON: A Self-Evolving, Fully-Adaptive Resource Arbitrator for Storage Systems	75
<i>Sandeep Uttamchandani, IBM Almaden Research Center; Li Yin, University of California, Berkeley; Guillermo A. Alvarez and John Palmer, IBM Almaden Research Center; Gul Agha, University of Illinois at Urbana-Champaign</i>	

Improving Filesystems

A Transactional Flash File System for Microcontrollers	89
<i>Eran Gal and Sivan Toledo, Tel-Aviv University</i>	
Analysis and Evolution of Journaling File Systems	105
<i>Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison</i>	
Comparison-Based File Server Verification	121
<i>Yuen-Lin Tan, Terrence Wong, John D. Strunk, and Gregory R. Ganger, Carnegie Mellon University</i>	

Thursday, April 14, 2005

Defending Against Attacks

Active Internet Traffic Filtering: Real-Time Response to Denial-of-Service Attacks 135
Katerina Argyraki and David R. Cheriton, Stanford University

Building a Reactive Immune System for Software Services 149
Stelios Sidiropoulos, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis, Columbia University

Attrition Defenses for a Peer-to-Peer Digital Preservation System 163
T.J. Giuli, Stanford University; Petros Maniatis, Intel Research; Mary Baker, Hewlett-Packard Labs; David S. H. Rosenthal, Stanford University; Mema Roussopoulos, Harvard University

Improving Data Movement

Peer-to-Peer Communication Across Network Address Translators 179
Bryan Ford, Massachusetts Institute of Technology; Pyda Srisuresh, Caymas Systems, Inc.; Dan Kegel

Maintaining High Bandwidth Under Dynamic Network Conditions 193
Dejan Kostić, Ryan Braud, Charles Killian, Erik Vandekeijf, James W. Anderson, Alex C. Snoeren, and Amin Vahdat, University of California, San Diego

Server Network Scalability and TCP Offload 209
Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum, Prashant Pradhan, and John Tracey, IBM T.J. Watson Research Center

Short Papers I and II: see p. v

Friday, April 15, 2005

Speeding Up Things

A Portable Kernel Abstraction for Low-Overhead Ephemeral Mapping Management 223
Khaled Elmeleegy, Anupam Chanda, and Alan L. Cox, Rice University; Willy Zwaenepoel, EPFL

Adaptive Main Memory Compression 237
Irina Chihaia Tuduce and Thomas Gross, ETH Zürich

Drive-Thru: Fast, Accurate Evaluation of Storage Power Management 251
Daniel Peek and Jason Flinn, University of Michigan

Large Systems

Itanium—A System Implementor's Tale 265
Charles Gray, University of New South Wales; Matthew Chapman and Peter Chubb, University of New South Wales and National ICT Australia; David Mosberger-Tang, Hewlett-Packard Labs; Gernot Heiser, University of New South Wales and National ICT Australia

Providing Dynamic Update in an Operating System 279
Andrew Baumann and Gernot Heiser, University of New South Wales and National ICT Australia; Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski, IBM T.J. Watson Research Center; Jeremy Kerr, IBM Linux Technology Center

SARC: Sequential Prefetching in Adaptive Replacement Cache 293
Binny S. Gill and Dharmendra S. Modha, IBM Almaden Research Center

Improving OS Components

- SLINKY: Static Linking Reloaded 309
Christian Collberg, John H. Hartman, Sridivya Babu, and Sharath K. Udupa, University of Arizona
- CLOCK-Pro: An Effective Improvement of the CLOCK Replacement 323
Song Jiang, Los Alamos National Laboratory; Feng Chen and Xiaodong Zhang, College of William and Mary
- Group Ratio Round-Robin: $O(1)$ Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems 337
Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoqiang Zheng, Columbia University

SHORT PAPERS

Thursday, April 14, 2005

Short Papers I

- A Hierarchical Semantic Overlay Approach to P2P Similarity Search 355
Duc A. Tran, University of Dayton
- A Parts-of-File File System 359
Yoann Padioleau and Olivier Ridoux, Campus Universitaire de Beaulieu
- BINDER: An Extrusion-Based Break-In Detector for Personal Computers 363
Weidong Cui and Randy H. Katz, University of California, Berkeley; Wai-tian Tan, Hewlett-Packard Laboratories
- Proper: Privileged Operations in a Virtualised System Environment 367
Steve Muir, Larry Peterson, and Marc Fiuczynski, Princeton University; Justin Cappos and John Hartman, University of Arizona
- AMP: Program Context Specific Buffer Caching 371
Feng Zhou, Rob von Behren, and Eric Brewer, University of California, Berkeley
- Automatic Synthesis of Filters to Discard Buffer Overflow Attacks: A Step Towards Realizing Self-Healing Systems 375
Zhenkai Liang, R. Sekar, and Daniel C. DuVarney, Stony Brook University

Short Papers II

- Facilitating the Development of Soft Devices 379
Andrew Warfield, Steven Hand, Keir Fraser, and Tim Deegan, University of Cambridge Computer Laboratory
- Implementing Transparent Shared Memory on Clusters Using Virtual Machines 383
Matthew Chapman and Gernot Heiser, University of New South Wales and National ICT Australia
- Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor 387
Ludmila Cherkasova and Rob Gardner, Hewlett-Packard Laboratories
- Fast Transparent Migration for Virtual Machines 391
Michael Nelson, Beng-Hong Lim, and Greg Hutchins, VMware, Inc.
- Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design 395
Alexandra Fedorova, Harvard University and Sun Microsystems; Margo Seltzer, Harvard University; Christopher Small and Daniel Nussbaum, Sun Microsystems
- Hyper-Threading Aware Process Scheduling Heuristics 399
James R. Bulpin and Ian A. Pratt, University of Cambridge Computer Laboratory

Index of Authors

Agha, Gul	75	Freimuth, Doug	209	Nethercote, Nicholas	17
Alvarez, Guillermo A.	75	Gal, Eran	89	Nieh, Jason	337
Anderson, James W.	193	Ganger, Gregory R.	121	Palmer, John	75
Appavoo, Jonathan	279	Gill, Binny S.	293	Peek, Daniel	251
Argyraki, Katerina	135	Giuli, T.J.	163	Prabhakaran, Vijayan	105
Arpaci-Dusseau, Andrea C.	105	Gray, Charles	265	Pradhan, Prashant	209
Arpaci-Dusseau, Remzi H.	105	Gross, Thomas	237	Ranganathan, Parthasarathy	61
Babu, Sridivya	309	Hartman, John H.	309	Rosenthal, David S. H.	163
Baker, Mary	163	Heiser, Gernot	265, 279	Roussopoulos, Mema	163
Baumann, Andrew	279	Hevia, Alejandro	45	Seward, Julian	17
Bhagwan, Ranjita	45	Hu, Elbert	209	Sharma, Ratnesh	61
Boyd, Stephen W.	149	Jiang, Song	323	Sidiroglou, Stelios	149
Braud, Ryan	193	Junqueira, Flavio	45	Snoeren, Alex C.	193
Caprita, Bogdan	337	Kegel, Dan	179	Sorin, Daniel J.	31
Chan, Wong Chun	337	Keromytis, Angelos D.	149	Srisuresh, Pyda	179
Chanda, Anupam	223	Kerr, Jeremy	279	Stein, Clifford	337
Chapman, Matthew	265	Killian, Charles	193	Strunk, John D.	121
Chase, Jeff	61	King, Samuel T.	1	Tan, Yuen-Lin	121
Chen, Feng	323	Kostić, Dejan	193	Toledo, Sivan	89
Chen, Peter M.	1	Krieger, Orran	279	Tracey, John	209
Cheriton, David R.	135	LaVoie, Jason	209	Udupa, Sharath K.	309
Chihaia Tuduce, Irina	237	Lebeck, Alvin R.	31	Uttamchandani, Sandeep	75
Chubb, Peter	265	Li, Tong	31	Vahdat, Amin	193
Collberg, Christian	309	Locasto, Michael E.	149	Vandekieft, Erik	193
Cox, Alan L.	223	Maniatis, Petros	163	Voelker, Geoffrey M.	45
Da Silva, Dilma	279	Marzullo, Keith	45	Wisniewski, Robert W.	279
Dunlap, George W.	1	Modha, Dharmendra S.	293	Wong, Terrence	121
Ellis, Carla S.	31	Moore, Justin	61	Yin, Li	75
Elmeleegy, Khaled	223	Mosberger-Tang, David	265	Zhang, Xiaodong	323
Flinn, Jason	251	Mraz, Ronald	209	Zheng, Haoqiang	337
Ford, Bryan	179	Nahum, Erich	209	Zwaenepoel, Willy	223

Index of Authors, Short Papers

Brewer, Eric	371	Hand, Steven	379	Pratt, Ian A.	399
Bulpin, James R.	399	Hartman, John	367	Ridoux, Olivier	359
Cappos, Justin	367	Heiser, Gernot	383	Sekar, R.	375
Chapman, Matthew	383	Hutchins, Greg	391	Seltzer, Margo	395
Cherkasova, Ludmila	387	Katz, Randy H.	363	Small, Christopher	395
Cui, Weidong	363	Liang, Zhenkai	375	Tan, Wai-tian	363
Deegan, Tim	379	Lim, Beng-Hong	391	Tran, Duc A.	355
DuVarney, Daniel C.	375	Muir, Steve	367	von Behren, Rob	371
Fedorova, Alexandra	395	Nelson, Michael	391	Warfield, Andrew	379
Fiuczynski, Marc	367	Nussbaum, Daniel	395	Zhou, Feng	371
Fraser, Keir	379	Padioleau, Yoann	359		
Gardner, Rob	387	Peterson, Larry	367		

Message from the Program Chair

Welcome to the 2005 USENIX Annual Technical Conference!

This conference has a long history of bringing together the best and brightest in the field, and producing interesting papers that withstand the test of time. I believe that this year will continue in this great tradition. At the same time, change is inevitable, and this year brings a number of changes to the conference.

The biggest change is moving the conference to April from its traditional June time frame. Being in the transition year of this change meant that submission deadlines, reviewing schedules, etc., all had to be adjusted. I would like to thank the Program Committee for their extreme dedication in managing to handle a very compressed review cycle with nary a complaint. After the initial reviews and many emails exchanged to prune the list, we were still left with over 50 papers to resolve in one long day in San Francisco on the Saturday before OSDI '04. Many of the PC members also had papers at OSDI '04 and/or WORLDS '04, so their dedication is much appreciated.

The second change is in the format, which now includes a short paper track. The goal of this track is to encourage people to submit earlier, shorter, or more experimental papers, and to bring about more of a “workshop” feel to USENIX. In total, we received 118 full paper submissions and 30 short paper submissions. Of these, we accepted 24 as full papers and 12 as short papers. The breadth of this year’s conference is quite large, from advances in the smallest embedded systems all the way to data-center and world-wide issues. We have papers in security, performance, file systems, operating systems, networking, and many other areas. I think few conferences of this quality can make similar claims.

A number of people need to be thanked for making all of this possible. I would like to acknowledge a few PC members for making my life much easier: Atul Adya for handling the short papers, Steve Gribble for handling all of my conflicts, and Lucy Cherkasova for really championing the short paper session. Also making the life of the Program Chair much easier is the whole USENIX organization, which deserves many thanks. In particular, I would like to give special thanks to Ellie Young and Jane-Ellen Long, who have often arranged things well before I even contemplated them. Their efforts and encouragement have made being chair relatively painless.

I hope you enjoy the 2005 USENIX Annual Technical Conference and will attend for years to come. Your participation is what makes all of this happen.

Vivek Pai, Princeton University
Program Chair

Debugging operating systems with time-traveling virtual machines

Samuel T. King, George W. Dunlap, and Peter M. Chen
University of Michigan

Abstract

Operating systems are difficult to debug with traditional cyclic debugging. They are non-deterministic; they run for long periods of time; they interact directly with hardware devices; and their state is easily perturbed by the act of debugging. This paper describes a time-traveling virtual machine that overcomes many of the difficulties associated with debugging operating systems. Time travel enables a programmer to navigate backward and forward arbitrarily through the execution history of a particular run and to replay arbitrary segments of the past execution. We integrate time travel into a general-purpose debugger to enable a programmer to debug an OS in reverse, implementing commands such as reverse breakpoint, reverse watchpoint, and reverse single step. The space and time overheads needed to support time travel are reasonable for debugging, and movements in time are fast enough to support interactive debugging. We demonstrate the value of our time-traveling virtual machine by using it to understand and fix several OS bugs that are difficult to find with standard debugging tools. Reverse debugging is especially helpful in finding bugs that are fragile due to non-determinism, bugs in device drivers, bugs that require long runs to trigger, bugs that corrupt the stack, and bugs that are detected after the relevant stack frame is popped.

1 Introduction

Computer programmers are all-too-familiar with the task of debugging complex software through a combination of detective work, guesswork, and systematic search. Tracking down a bug generally starts with running a program until an error in the program manifests as a fault. The programmer¹ then seeks to start from the fault (the manifestation of the error) and work backward to the cause of the fault (the programming error itself). Cyclic debugging is the classic way to work backward toward the error. In cyclic debugging, a programmer uses a debugger or output statements to examine the state of the program at a given point in its execution. Armed with

¹In this paper, “programmer” refers to the person debugging the system, and “debugger” refers to the programming tool (e.g., `gdb`) used by the programmer to examine and control the program.

this information, the programmer then re-runs the program, stops it at an earlier point in its execution history, examines the state at this point, then iterates.

Unfortunately, this classic approach to debugging is difficult to apply when debugging operating systems. Many aspects of operating systems make them difficult to debug: they are non-deterministic; they run for long periods of time; the act of debugging may perturb their state; and they interact directly with hardware devices.

First, operating systems are non-deterministic. Their execution is affected by non-deterministic events such as the interleaving of multiple threads, interrupts, user input, network input, and the perturbations of state caused by the programmer who is debugging the system. This non-determinism makes cyclic debugging infeasible because the programmer cannot re-run the system to examine the state at an earlier point.

Second, operating systems run for long periods of time, such as weeks, months, or even years. Re-running the system in cyclic debugging would thus be infeasible even if the OS were completely deterministic.

Third, the act of debugging may perturb the state of the operating system. The converse is also true: a misbehaving operating system may corrupt the state of the debugger. These interactions are possible because the operating system is traditionally the lowest level of software on a computer, so the debugger’s code and data is not isolated from the OS (unless the debugger uses specialized hardware such as an in-circuit emulator). Even remote kernel debuggers depend on some basic functionality in the debugged OS, such as reading and writing memory locations, setting and handling breakpoints, and communicating with the remote debugger (e.g., through the serial line). Using this basic functionality may be impossible on a sick OS. A debugger also needs assistance from the OS to access hardware devices, and this functionality may not work on a sick OS.

Finally, operating systems interact directly with hardware devices. Devices are sources of non-determinism that hinder cyclic debugging; they return data and generate interrupts that may change between runs. Devices may also fail due to timing dependencies if a programmer pauses during a debugging session.

In this paper, we describe how to use *time-traveling virtual machines* to overcome many of the difficulties as-

sociated with debugging operating systems. By virtual machine, we mean a software-implemented abstraction of a physical machine that is at a low-enough level to run an operating system. Running the OS inside a virtual machine enables the programmer to stand outside the OS being debugged. From this vantage point, the programmer can use a debugger to examine and control the execution of the OS without perturbing its state.

By time travel, we mean the ability to navigate backward and forward arbitrarily through the execution history of a particular run and to replay arbitrary segments of the past execution. For example, if the system crashed due to an errant pointer variable, time travel would allow the programmer to go back to the point when that pointer variable was corrupted; it would also allow the programmer to fast-forward again to the crash point. Time-traveling virtual machines allow a programmer to replay a prior point in the execution exactly as it was executed the first time. The past is immutable in our model of time travel; this ensures that there is only a single execution history, rather than a branching set of execution histories. As with cyclic debugging, the goal of time travel is to enable the programmer to examine the state of the OS at prior points in the execution. However, unlike cyclic debugging, time travel works in the presence of non-determinism. Time travel is also more convenient than classic cyclic debugging because it does not require the entire run to be repeated.

In this paper, we describe the design and implementation of a time-traveling virtual machine (TTVM) for debugging operating systems. We integrate time travel into a general-purpose debugger (`gdb`) for our virtual machine, implementing commands such as reverse step (go back to the last instruction that was executed), reverse breakpoint (go back to the last time an instruction was executed), and reverse watchpoint (go back to the last time a variable was modified).

The space and time overhead needed to support time travel is reasonable for debugging. For three workloads that exercise the OS intensively, the logging needed to support time travel adds 3-12% time overhead and 2-85 KB/sec space overhead. The speed at which one can move backward and forward in the execution history depends on the frequency of checkpoints in the time region of interest. TTVM is able to insert additional checkpoints to speed up these movements or delete existing checkpoints to reduce space overhead. After adding checkpoints to a region of interest, TTVM allows a programmer to move to an arbitrary point within the region in about 12 seconds.

The following real-life example clarifies what we mean by debugging with time-traveling virtual machines and illustrates the value of debugging in this manner. The error we were attempting to debug was triggered

when the guest kernel attempted to call a NULL function pointer. The error had corrupted the stack, so standard debugging tools were unable to traverse the call stack and determine where the invalid function call had originated. Using the TTVM reverse single step command, we were able easily to step back to where the function invocation was attempted and examine the state of the virtual machine at that point.

The contributions of this paper are as follows. TTVM is the first system that provides practical reverse debugging for long-running, multi-threaded programs such as an operating system. We show how to provide this capability at reasonable time and space overhead through techniques such as virtual-machine replay, checkpointing, logging disks, and running native device drivers inside a virtual machine. We also show how to integrate time travel in a debugger to enable new reverse debugging commands. We illustrate the usefulness of reverse debugging for operating systems through anecdotal experience and generalize about the types of situations in which reverse debugging is particularly helpful.

2 Virtual machines

A virtual machine is a software abstraction of a physical machine [12]. The software layer that provides this abstraction is called a virtual machine monitor (VMM). An operating system can be installed and run on a virtual machine as if it were running on a physical machine. Such an OS is called a “guest” OS to distinguish it from an OS that may be integrated into the VMM itself (which is called the “host” OS).

Several features of virtual machines make them attractive for our purposes. First, because the VMM adds a layer of software below the guest OS, it provides a protected substrate in which one can add new features. Unlike traditional kernel debugging, these new features will continue to work regardless of how sick the guest OS becomes; the guest OS cannot corrupt or interfere with the debugging functionality. We use this substrate to add traditional debugging capabilities such as setting breakpoints and reading and writing memory locations. We also add non-traditional debugging features such as logging and replaying non-deterministic inputs and saving and restoring the state of the virtual machine.

Second, a VMM allows us to run a general-purpose, full-featured debugger on the same physical machine as the OS being debugged without perturbing the debugged OS. Compared to traditional kernel debuggers, virtual machines enable more powerful debugging capabilities (e.g., one can read the virtual disk) with no perturbation of or dependency on the OS being debugged. It is also more convenient to use than a remote debugger because it does not require a second physical machine.

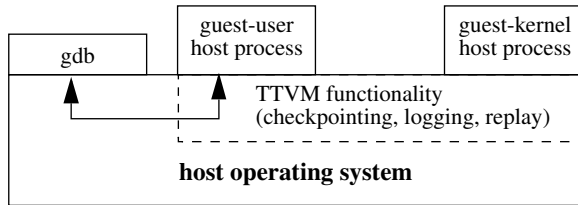


Figure 1: System structure: UML runs as two user processes on the host Linux OS, the *guest-kernel host process* and the *guest-user host process*. TTVM’s ability to travel forward and back in time is implemented by modifying the host OS. We extend *gdb* to make use of this time traveling functionality. *gdb* communicates with the guest-kernel host process via a remote serial protocol.

Finally, a VMM offers a narrow and well-defined interface: the interface of a physical machine. This interface makes it easier to implement the checkpointing and replay features we add in this paper, especially compared to the relatively wide and complex interface offered by an operating system to its application processes. The state of a virtual machine is easily identified as the virtual machine’s memory, disk, and registers and can thus be saved and restored easily. Replay is easier to implement in a VMM than an operating system because the VMM exports an abstraction of a uniprocessor virtual machine (assuming a uniprocessor physical machine), whereas an OS exports an abstraction of a virtual multiprocessor to its application processes.

The VMM used in this paper is User-Mode Linux (UML) [8], modified to support host device drivers in the guest OS. UML is implemented as a kernel modification to a host Linux OS (Figure 1)². The virtual machine runs as two user processes on the host OS: one host process (the *guest-kernel host process*) runs all guest kernel code, and one host process (the *guest-user host process*) runs all guest user code. The guest-kernel host process uses the Linux *ptrace* facility to intercept system calls and signals generated by the guest-user host process. The guest-user host process uses UML’s *skas*-extension to the host Linux kernel to switch quickly between address spaces of different guest user processes.

UML’s VMM exports a para-virtualized architecture that is similar but not identical to the host hardware [28]. The guest OS in UML, which is also Linux, must be ported to run on top of this virtual architecture. Each piece of virtual hardware in UML is emulated with a host service. The guest disk is emulated by a raw disk partition on the host; the guest memory is emulated by a memory-mapped file on the host; the guest network

card is emulated by a host TUN/TAP virtual Ethernet driver; the guest MMU is emulated by calls to the host *mmap* and *mprotect* system calls; guest timer and device interrupts are emulated by host *SIGALRM* and *SIGIO* signals; the guest console is emulated by standard output. The guest Linux’s architecture-dependent layer uses these host services to interact with the virtual hardware.

Using a para-virtualized VMM [28] such as UML raises the issue of fidelity: is the guest OS similar enough to an OS that runs on the hardware (i.e., a host OS) that one can track down a bug in a host OS by debugging the guest OS? The answer depends on the specific VMM: as the para-virtualized architecture diverges from the hardware architecture, a guest OS that runs on the para-virtualized architecture diverges from the host OS, and it becomes less likely that a bug in the host OS can be debugged in the guest OS. Timing-dependent bugs may also manifest differently when running an OS on a virtual machine than when running on hardware.

UML’s VMM is similar enough to the hardware interface that most code is identical between a host OS and a guest OS. The differences between the host OS and guest OS are isolated to the architecture-specific code, and almost all these differences are in device drivers. Not including device driver code, 92% of the code (measured in lines of *.c* and *.S* files) are identical between the guest and host OS. Because many OS bugs are in device drivers [7], we added the capability to UML to use unmodified real device drivers in the guest OS to drive devices on the host platform (Section 3.2)[17, 11]. This makes it possible to debug problems in real device drivers with our system, and we have used our system to find, fix, and submit a patch for a bug in the host OS’s USB serial driver. With our extension to UML, 98% of the host OS code base (including device drivers) can be debugged in the guest OS. Applying the techniques in this paper to a non para-virtualized VMM such as VMware would enable reverse debugging to work for any host OS bug.

Running an OS inside a virtual machine incurs overhead. We measured UML’s virtualization overhead as 0% for the POV-ray ray tracer (a compute-intensive workload), 76% for a build of the Linux kernel (a system-call intensive workload which is expensive to virtualize [15]), and 15% for SPECweb99 (a web-server workload). This overhead is acceptable for debugging (in fact, UML is used in production web hosting environments). If lower overhead is needed, the ideas in this paper can be applied to faster virtual machines such as Xen [3] (3% overhead for a Linux kernel build), UMLinux/FAUmachine [15] (35% overhead for a Linux kernel build), or a hardware-supported virtual machine such as Intel’s upcoming Vanderpool Technology.

²We use the *skas* (separate kernel address space) version of UML, which requires a patch of the host kernel.

3 Time-traveling virtual machines

A time-traveling virtual machine should have two capabilities. First, it should be able to reconstruct the complete state of the virtual machine at any point in a run, where a run is defined as the time from when the virtual machine was powered on to the last instruction it executed. Second, it should be able to start from any point in a run and from that point replay the same instruction stream that was executed during the original run from that point. This section describes how TTVM achieves these capabilities through a combination of logging, replay, and checkpointing.

3.1 Logging and replaying a VM

The foundational capability in TTVM is the ability to replay a run from a given point in a way that matches the original run instruction for instruction. Replay causes the virtual machine to transition through the same states as it went through during the original run; hence replay enables one to reconstruct the complete state of the virtual machine at any point in the run. TTVM uses the ReVirt logging/replay system to provide this capability [9]. This section briefly summarizes how ReVirt logs and replays the execution of a virtual machine.

A virtual machine can be replayed by starting from a checkpoint, then replaying all sources of non-determinism [5, 9]. For UML, the sources of non-determinism are external input from the network, keyboard, and real-time clock and the timing of virtual interrupts. The VMM replays network and keyboard input by logging the calls that read these devices during the original run and regenerating the same data during the replay run. Likewise, we configure the CPU to cause reads of the real-time clock to trap to the VMM, where they can be logged or regenerated.

To replay a virtual interrupt, ReVirt logs the instruction in the run at which it was delivered and re-delivers the interrupt at this instruction during replay. This point is identified uniquely by the number of branches since the start of the run and the address of the interrupted instruction [19]. ReVirt uses a performance counter on the Intel Pentium 4 CPU to count the number of branches during logging, and it uses the same performance counter and instruction breakpoints to stop at the interrupted instruction during replay. Replaying interrupts enables ReVirt to replay the scheduling order of multi-threaded guest operating systems and applications, as long as the VMM exports the abstraction of a uniprocessor virtual machine [22]. Researchers are investigating ways to support replay on multiprocessors [29].

3.2 Host device drivers in the guest OS

In general, VMMs export a limited set of virtual devices. Some VMMs export virtual devices that exist in hardware (e.g., VMware Workstation exports an emulated AMD Lance Ethernet card); others (like UML) export virtual devices that have no hardware equivalent. Exporting a limited set of virtual devices to the guest OS is usually considered a benefit of virtual-machine systems, because it frees guest OSs from needing device drivers for myriad host devices [26]. However, when using virtual machines to debug operating systems, the limited set of virtual devices prevents programmers from using and debugging drivers for real devices; programmers can only debug the architecture-independent portion of the guest OS. There are two ways to address this limitation and enable the programmer to run and debug real device drivers in a guest OS. With both strategies, real device drivers can be included in the guest OS without being modified or re-compiled.

The first way to run a real device driver in the guest OS is for the VMM to provide a software emulator for that device. The device driver issues the normal set of I/O instructions: IN/OUT instructions, memory-mapped I/O, DMA commands, and interrupts. The VMM traps these privileged instructions and forwards them to/from the software device emulator. With this strategy, ReVirt can log and replay device driver code in the same way it logs and replays the rest of the guest OS. If one runs the VMM's software device emulator above ReVirt's logging system (and above the checkpoint system described in Section 3.3), ReVirt will guide the emulator and device driver code through the same instruction sequence during replay as they executed during logging. While this first strategy fits in well with the existing ReVirt system, it only works if one has an accurate software emulator for the device whose driver one wishes to debug.

We modified UML to provide a second way to run real device drivers in the guest OS, which works even when no software emulator exists for the device of interest. With this strategy, the VMM traps and forwards the privileged I/O instructions and DMA requests issued by the guest OS device driver to the actual hardware. The programmer specifies which devices UML can access, and the VMM enforces the proper I/O port space and memory access for the device.

This second strategy requires extensions to enable ReVirt to log and replay the execution of the device driver. Whereas the first strategy placed the device emulator above the ReVirt logging layer, the second strategy forwards driver actions to the actual hardware device. Because this device may not be deterministic, ReVirt must log any information sent from the device to the driver. Specifically, ReVirt must log and replay the data returned

by IN instructions, memory-mapped I/O instructions, and DMA memory loads. To avoid confusing the device, ReVirt suppresses output to the device during replay.

The VMM must also be modified to support running real device drivers in the guest OS. Supporting x86 IN/OUT instructions is straightforward since they are privileged and naturally trap to the VMM. After receiving a trap from an IN/OUT instruction, TTVM verifies the port address and forwards the instruction to the device. After the instruction is executed, TTVM transparently passes the result back to the guest. Like IN/OUT instructions, interrupt handling requires few modifications. UML already uses signals in place of hardware interrupts, so when the VMM receives an interrupt from a device, it is forwarded to the guest using signals.

To support memory-mapped I/O and DMA, we augmented the guest OS's memory-mapping and DMA allocation routines to request access to the host's physical memory by issuing system calls to the host. For memory-mapped I/O, the guest OS asks the host to map the desired I/O region into the guest OS's address space; for DMA, the guest OS asks the host to allocate physical memory suitable for DMA transfers. These actions are necessary since the guest is controlling a real device. However, because ReVirt must log all loads from the resulting virtual address range, the guest cannot have unchecked access to the newly allocated resources. As a result, TTVM uses page protections to trap all interactions with the allocated virtual memory range. Upon receiving a trap, TTVM emulates all guest driver loads and stores that interact with memory-mapped I/O space or DMA memory range. This provides sufficient opportunity to log and replay all interactions between the guest driver and the device.

One shortcoming of this approach is that the extra traps and logging operations slows loads and stores to memory-mapped I/O space and DMA memory. In practice, this slowdown is minimized since most bulk transfers are implemented using x86 `repeat string` instructions, so bulk transfers cause only a single trap. We experienced no noticeable slowdown as a result of using this mechanism. For example, a guest USB serial port driver can operate at full speed, and the guest OS sound-card driver can play an MP3 music clip and record audio in real-time.

Allowing the guest device driver to initiate DMA transfers allows the guest OS to potentially corrupt host memory, since the device can access all of the host's physical memory [17]. The programmer who is worried about this possibility can interpret DMA setup commands and deny access to memory outside the intended range. Some recent processors, such as AMD's Opteron, provide an I/O MMU which can be used to restrict accesses to the intended memory range.

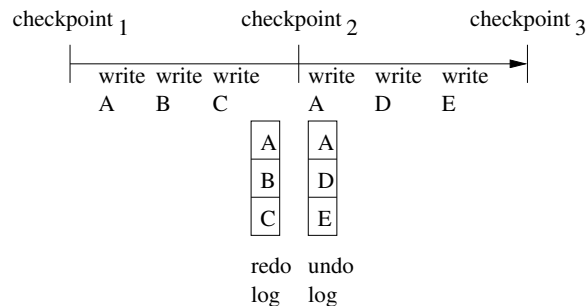


Figure 2: Checkpoints of the memory pages are represented as undo and redo logs. The figure shows the redo and undo logs that would result for checkpoint₂ for the given sequence of writes to memory pages. The same technique is used to store the changes to the *mappings* of guest→host disk blocks.

3.3 Checkpointing for faster time travel

Logging and replaying a virtual machine from a single checkpoint at the beginning of the run is sufficient to recreate the state at any point in the run from any other point in the run. However, logging and replay alone is not sufficient to recreate this state *quickly* because the virtual machine must re-execute each instruction from the beginning to the desired point, and this period may span many days. To accelerate time travel over long periods, TTVM takes periodic checkpoints while the virtual machine is running [23] (ReVirt started only from a disk checkpoint of a powered-off virtual machine).

The simplest way to checkpoint the virtual machine is to save a complete copy of the state of the virtual machine. This state is comprised of the CPU registers, the virtual machine's physical memory, the virtual disk, and any state in the VMM or host kernel that affects the execution of the virtual machine. For UML, this host kernel state includes the address space mappings for the guest-user host process and the guest-kernel host process, the state of open host file descriptors, and the registration of various signal handlers (analogous to the interrupt descriptor table on real hardware).

Saving a complete copy of the virtual-machine state is simple but inefficient. We use copy-on-write and versioning to reduce the space and time overhead of checkpointing for both memory pages and disk blocks.

We use copy-on-write on memory pages to save only those pages that have been modified since the last checkpoint. Starting with the memory contents at a current point, the memory state can be restored back to a prior checkpoint by restoring the memory pages in the undo log. The memory undo log at checkpoint_{*n*} contains the set of memory pages that have been modified be-

tween checkpoint_{*n*} and checkpoint_{*n*+1}, with the values of the pages in the undo log being those at checkpoint_{*n*} (Figure 2). Analogously, TTVM uses a redo log of memory to enable a programmer to move forward in time to a future checkpoint. The memory redo log at checkpoint_{*n*} contains the set of memory pages that have been modified between checkpoint_{*n*-1} and checkpoint_{*n*}, with the values of these memory pages again being those at checkpoint_{*n*} (Figure 2). If a memory page is modified during two successive checkpoint intervals, the memory undo and redo logs for the checkpoint between these two intervals will contain the same values for that memory page. TTVM detects this case and shares such data between the undo and redo logs. E.g., in Figure 2, page A's data is shared between checkpoint₂'s undo and redo logs.

We use similar logging techniques for disk, but we add an extra level of indirection to avoid copying disk blocks into the undo and redo logs. The extra level of indirection is implemented by saving multiple versions of each guest disk block [25] and maintaining, in memory, the current mapping from guest disk blocks to host disk blocks. The first time a guest disk block is written after a checkpoint, TTVM writes the data to a free host disk block and updates the mapping from guest to host disk blocks to point to the new host disk block. This strategy saves copying the before-image of the guest disk block into the undo log of the prior checkpoint, and it saves copying the after-image of the disk block into the redo log of the next checkpoint. The undo and redo logs need store only the changes to the guest→host disk block map. Changes to the map are several orders of magnitude smaller than the disk block data and can be write buffered in non-volatile RAM to provide persistence if the host crashes.

3.4 Time traveling between points of a run

TTVM enables a programmer to travel arbitrarily backward and forward in time through a run. Time traveling between points in a run requires a combination of restoring to a checkpoint and replay. To travel from point A to point B, TTVM first restores to the checkpoint that is prior to point B (call this checkpoint_{*n*}). TTVM then replays the execution of the virtual machine from checkpoint_{*n*} to point B. The more frequently checkpoints are taken, the smaller the expected duration of the replay phase of time travel.

Restoring to checkpoint_{*n*} requires several steps. TTVM first restores the copy saved at checkpoint_{*n*} of the virtual machine's registers and any state in the VMM or host kernel that affects the execution of the virtual machine. Restoring the memory image and guest→host disk block map to the values they had at checkpoint_{*n*} makes use of the data stored in the undo logs if moving backward in time, or redo logs when moving for-

ward in time. Consider how to move from a point after checkpoint_{*n*+2} backward to checkpoint_{*n*} (restoring to a checkpoint in the future uses the redo log in an analogous manner). TTVM first restores the memory pages and disk block map entries from the undo log at checkpoint_{*n*}. It then examines the undo log at checkpoint_{*n*+1} and restores any memory pages and disk block map entries that were not restored by the undo log at checkpoint_{*n*}. Finally, TTVM examines the undo log at checkpoint_{*n*+2} and restores any memory pages and disk block map entries that were not restored by the undo logs at checkpoint_{*n*} or checkpoint_{*n*+1}. Applying the logs in this order ensures that each memory page is written at most once.

3.5 Adding and deleting checkpoints

An initial set of checkpoints are taken during the original, logged run. TTVM supports the ability to add or delete checkpoints from this original set. At any time, the user may choose to delete existing checkpoints to free up space. While replaying a portion of a run, a programmer may choose to supplement the initial set of checkpoints to speed up anticipated time-travel operations. This section describes how to manipulate the undo and redo logs of the memory pages when adding or deleting a checkpoint. The undo and redo logs for the guest→host disk block map are maintained in exactly the same manner.

Adding a new checkpoint can be done when the programmer is replaying a portion of a run from a checkpoint (say, checkpoint₁). TTVM can add a new checkpoint₂ at the current point of replay (between existing checkpoint₁ and checkpoint₃) by creating the undo and redo logs for checkpoint₂. TTVM identifies the memory pages to store in checkpoint₂'s redo log by maintaining a list of the memory pages that are modified since the system started replaying at checkpoint₁, just as it does during logging to support the copy-on-write undo log. TTVM conservatively identifies the memory pages to store in checkpoint₂'s undo log as the same set of pages in checkpoint₁'s undo log, but with the values at the current point of replay. TTVM could remove memory pages from checkpoint₁'s undo log that were not written between checkpoint₁ and checkpoint₂, but this is not needed for correctness and TTVM does not currently include this optimization. It is difficult to remove extra pages from checkpoint₃'s redo log without executing through to checkpoint₃, because knowing which pages to remove would require knowing the time of the last modification to the page, and this would require trapping all modifications to all memory pages.

Deleting an existing checkpoint (presumably to free up space for a new checkpoint) can be done during the original logging run or when the programmer is

replaying a portion of a run. TTVM goes through two steps to delete checkpoint₂ (between checkpoint₁ and checkpoint₃). TTVM first moves the pages in checkpoint₂'s undo log to checkpoint₁'s undo log. A page that already exists in checkpoint₁'s undo log takes precedence over a page from checkpoint₂'s undo log. Similarly, TTVM moves the pages in checkpoint₂'s redo log to checkpoint₃'s redo log. A page that already exists in checkpoint₃'s redo log takes precedence over a page from checkpoint₂'s redo log.

3.6 Expected usage model

We expect programmers to use TTVM in three phases. Throughout each phase, TTVM will take checkpoints at a specified frequency (the default is every 25 seconds). In phase 1, the programmer runs a test to trigger an error. This phase may last a long time (hours or days). As we will see in Section 5, taking checkpoints every 25 seconds adds less than 4% time overhead, so it is reasonable to leave checkpointing on even during long runs.

For long runs, the space needed to store the undo/redo logs for all checkpoints will build up and TTVM will be forced to delete some checkpoints. By default, TTVM keeps more checkpoints for periods near the current time than for periods farther in the past; this policy assumes that periods in the near past are likely to be the ones of interest during debugging. TTVM chooses checkpoints to delete by fitting them to a distribution in which the distance between checkpoints increases exponentially as one goes farther back in time [4].

In phase 2, the programmer attaches the debugger, switches the system from logging to replay, and prepares to debug the error. To speed up later time-travel operations, programmers can specify a shorter interval between checkpoints (say, every 10 seconds), then replay the portion of the run they expect to debug (say, a 10 minute interval). As in phase 1, TTVM will keep checkpoints according to an exponential distribution that favors checkpoints close to the current (replaying) time.

In phase 3, the programmer debugs the error by time-traveling forward and backward through the run. We next describe new debugging commands that allow a programmer to navigate conveniently through the run.

4 TTVM-aware gdb

In this section, we discuss how to integrate the time traveling capability of TTVM into a debugger (gdb). We first introduce the new reverse debugging commands and discuss how they are implemented. We then describe how to manage the interaction of time traveling with the state changes generated by gdb. Finally, we describe

how our prototype implements communication between gdb and TTVM.

4.1 Time travel within gdb

In addition to the standard set of commands available to debuggers, TTVM allows gdb to restore prior checkpoints, replay portions of the execution, and examine arbitrary past states. A promising application of these technique is providing the illusion of virtual-machine reverse execution.

Reverse execution, when applied to debugging, provides the functionality standard debuggers are often trying to approximate. For example, a kernel may follow an errant pointer, read an unintended data structure, and crash. Using a standard debugger, the programmer can gain control when the crash occurs. A common approach at this point is to traverse up the call stack. This approximates reverse execution because it allows the programmer to see the partial state of function invocations that occurred before the crash. However, it only allows the programmer to see variables stored on the stack, and it only shows the values for those variables at the time of each function invocation. Another approach is to re-run the system with a watchpoint set on the pointer variable. However, this approach works only if the bug is deterministic. Also, the programmer may have to step through many watchpoints to get to the modification of interest. Ideally, the programmer would like to go to the *last* time the pointer was modified. However, current debugging commands only allow the programmer to go to the *next* modification of the pointer.

To overcome this deficiency, we add a new command to gdb called `reverse continue`. `reverse continue` takes the virtual machine back to a *previous* point, where the point is identified by the reverse equivalents of forward breakpoints, watchpoints, and steps. In the example above, the programmer could set a watchpoint on the pointer variable and issue the `reverse continue` command. After executing this command, the debugger would return control to the programmer at the *last* time the variable was modified. This jump backward in time restores all virtual-machine state, so the programmer could then use standard gdb commands to gather further information.

`reverse continue` is implemented using two execution passes (Figure 3). In the first pass, TTVM restores a checkpoint that is earlier in the execution and replays the virtual machine until the current location is reached again. During the replay of the first pass, gdb receives control on each trap caused by gdb commands issued by the programmer (e.g., breakpoints, watchpoints, steps). gdb keeps a list of these traps and, when the first pass is over, allows the programmer to choose a trap to

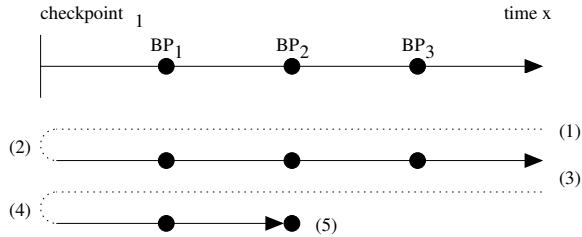


Figure 3: Reverse continue uses two execution passes. The programmer calls `reverse continue` at time x . In the first pass, (1) TTVM restores `checkpoint1`, then (2) replays execution until time x . Along the way, TTVM makes note of breakpoints `BP1`, `BP2`, and `BP3`. When time x is reached, the programmer sees a list of these breakpoints and selects one to go back to. In the example shown here, the programmer selects `BP2`. In the second pass, TTVM again (3) restores `checkpoint1` and (4) replays execution, but this time TTVM stops at breakpoint `BP2` and returns control to the programmer (5).

time travel back to. During the second pass, `gdb` again restores the same checkpoint and replays. When the selected trap is encountered during the second pass, `gdb` returns control to the programmer.

This approach is general enough that it provides reverse versions to *all* `gdb` commands. For example, the programmer can set instruction breakpoints, conditional breakpoints, data watchpoints, or single steps (or combinations thereof), and the `reverse continue` command keeps track of all resulting traps and allows the programmer to go back to any of them. We have found each of these reverse commands useful in our kernel debugging (Section 6).

We found `reverse step` to be a particularly useful command (`reverse step` goes back a specified number of instructions). This command is particularly useful because it tracks instructions executed in guest kernel mode regardless of the kernel entry point. For example, if `gdb` has control inside a guest interrupt handler, and the interrupt occurred while the guest kernel was running, `reverse step` can go backward to determine which guest kernel instruction was preempted. We implemented an optimized version of the `reverse step` command because it is used so frequently and because the unoptimized version generates an inordinate number of traps. On x86, `gdb` uses the CPU’s `trap` flag to single step forward. `reverse step` also uses the `trap` flag, but doing so naively would generate a trap to `gdb` on each instruction replayed from the checkpoint. To reduce the number of traps caused by `reverse step`, we wait to set the `trap` flag during each pass’s replay until the system is near the current point. Our current im-

plementation defines “near” to be within one system call of the current point, but one could easily define “near” to be within a certain number of branches.

Finally, we implemented a `goto` command that a programmer can use to jump to an arbitrary time in the execution, either behind or ahead of the current point. Our current prototype defines time in a coarse-grained manner by counting guest system calls, but it is possible to define time by logging the real-time clock, or by counting branches. `goto` is most useful when the programmer is trying to find a time (possibly far from the current point) when an error condition is present.

4.2 TTVM/debugger interactions

Time traveling must affect debugging state (e.g., the set of breakpoints) differently from how it affects other virtual-machine state. Time-travel operations change virtual-machine state but should preserve debugging state. For example, if the programmer sets a breakpoint and executes `reverse continue`, the breakpoint must be unperturbed by the checkpoint restoration so that it can trap to `gdb` during the replay passes. Unfortunately, `gdb` mingles debugging state and virtual-machine state. For example, `gdb` implements software breakpoints by inserting `breakpoint` instructions directly into the code page of the process being debugged.

To enable special treatment of debugging state, TTVM tracks all modifications `gdb` makes to the virtual state. This allows TTVM to make debugging state persistent across checkpoint restores by manually restoring the debugging state after the checkpoint is restored. In addition, TTVM removes any modifications caused by the debugger before taking a checkpoint, so that the checkpoint includes only the original virtual-machine state.

4.3 TTVM on guest applications

While the focus of this paper is using TTVM to debug guest kernels, TTVM can also be used to debug multi-threaded guest applications. In order to debug guest applications, TTVM must be able to detect the currently running guest process from within the host kernel.

Detecting the current guest process is important because UML multiplexes a single host process address space between all guest application processes. Because of this multiplexing, TTVM must detect which guest process is currently occupying the host process address space before applying any modifications needed for debugging. For example, if TTVM tries to set a breakpoint in process A, but process B is currently running, TTVM must wait until process A is switched back in before applying any changes. Otherwise, process B will incorrectly trigger the breakpoint.

To determine the current guest process, TTVM must understand guest kernel task structs. Fortunately, the guest kernel stack pointer is known within the host kernel, and the current guest application pid is in a well-known location relative to the stack pointer.

With these enhancements, TTVM enables programmers to use reverse debugging commands for debugging guest applications.

4.4 Reverse gdb implementation

`gdb` and TTVM communicate via the `gdb` remote serial protocol (Figure 1). The remote serial protocol between `gdb` and TTVM is implemented in a host kernel device driver. `gdb` already understands the remote serial protocol and so need not be modified. The host kernel device driver receives the low-level remote protocol commands and reads/writes the state of the virtual machine on behalf of the debugger. These reads and writes are transparent to the virtual machine: neither the execution or replay of the virtual machine is affected (unless the guest kernel reads state that has been modified by `gdb`).

Although `gdb` did not have to be modified to understand the remote serial protocol, it did have to be extended to implement the new reverse commands. This provided complete integration of the new reverse commands inside the familiar `gdb` environment.

5 Performance

In this section, we measure the time and space overhead of TTVM and the time to execute time-travel operations. Since debugging is dominated by human think time, our main goal in this section is only to verify that the overhead of TTVM is reasonable.

All measurements are carried out on a uniprocessor 3 GHz Pentium 4 with 1 GB of memory and a 120 GB Hitachi Deskstar GXP disk. The host OS is Linux 2.4.18 with the `skas` extensions for UML and TTVM modifications. The guest OS is the UML port of Linux 2.4.20 and includes host drivers for the USB and soundcard devices. We configure the guest to have 256 MB of memory and a 5 GB disk, which is stored on a host raw disk partition. Both host and guest file systems are initialized from a RedHat 9 distribution. All results represent the average of at least 5 trials.

We measure three guest workloads: SPECweb99 using the Apache web server, three successive builds of the Linux 2.4 kernel (each of the three builds executes `make clean; make dep; make bzImage`), and the PostMark file system benchmark [14].

We first measure the time and space overhead of the logging needed to support replay. Checkpointing is disabled for this set of measurements. Running these work-

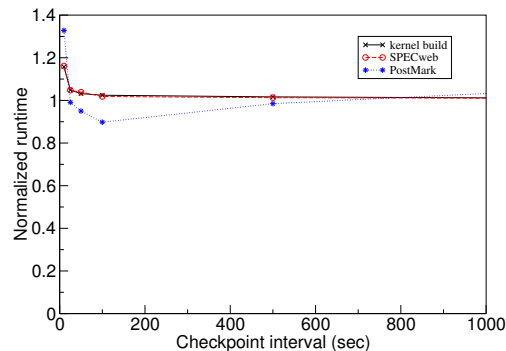


Figure 4: The effect of checkpointing on running time. Running times are normalized to running the workload without any checkpoints (1027 seconds for kernel build, 1135 seconds for SPECweb, 1114 seconds for PostMark). Overhead is low even for very short checkpoint intervals of 10 seconds.

loads on TTVM with logging adds 12% time overhead for SPECweb99, 11% time overhead for kernel build, and 3% time overhead for PostMark, relative to running the same workload in UML on standard Linux (with `skas`). The space overhead of TTVM needed to support logging is 85 KB/sec for SPECweb99, 7 KB/sec for kernel build, and 2 KB/sec for PostMark. These time and space overheads are easily acceptable for debugging.

Replay occurs at approximately the same speed as the logged run. For the three workloads, TTVM takes 1-3% longer to replay than it did to log. For workloads with idle periods, replay can be much faster than logging because TTVM skips over idle periods during replay.

We next measure the cost of enabling checkpointing. Figures 4 and 5 show how the time and space overheads of checkpointing vary with the interval between checkpoints. Taking checkpoints adds a small amount of time overhead and a modest amount of space overhead. Taking checkpoints every 25 seconds adds less than 4% time overhead and 2-6 MB/s space overhead. Even taking checkpoints as frequently as every 10 seconds is feasible for moderate periods of time, adding 15-27% time overhead and 4-7 MB/s space overhead.

This low space and time overhead is due to using undo/redo logs for memory data and logging for disk data. In particular, logging new versions of guest disk blocks rather than overwriting the old versions allowed us to perform checkpointing with negligible extra I/O (a checkpoint contains only the changes to the guest→host disk block map). Surprisingly, taking checkpoints more frequently sometimes improves PostMark's running time. The reason for this is how we allocate disk blocks. A guest disk block is assigned to a new host disk block only on the first time the guest disk block is writ-

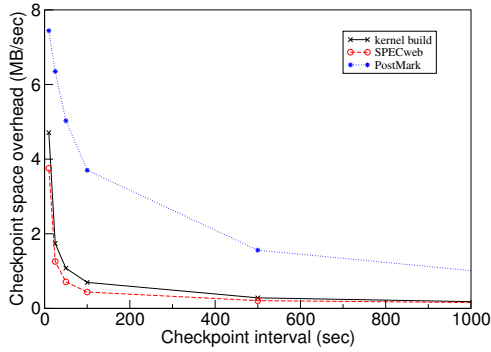


Figure 5: Space overhead of checkpoints. For long runs, programmers will cap the maximum space used by checkpoints by deleting selected checkpoints.

ten after a checkpoint. More frequent checkpoints thus cause the disk block allocation to resemble a pure logging disk, which improved the spatial locality for writes for PostMark.

Because checkpointing adds little time overhead, it is reasonable to perform long debugging runs while checkpointing relatively often (say, every 25 seconds). The space overhead of checkpoints over long runs will be capped typically at a maximum size, which causes TTVM to delete checkpoints according to its default exponential-thinning policy [4].

Next we consider the speed of moving forward and backward through the execution of a run. As described in Section 3.4, time travel takes two steps: (1) restoring to the checkpoint prior to the target point and (2) replaying the execution from this checkpoint to the target point. Figure 6 shows the time to restore a checkpoint as a function of the distance from the current point to a prior or future checkpoint. We used a checkpoint interval of 25 seconds and spanned the run with about 40 checkpoints. Moving to a checkpoint farther away takes more time because TTVM must examine and restore more undo/redo logs for memory pages and the disk block map. Recall that each unique memory page is written at most once, even when restoring to a point that is many checkpoints away. Hence the maximum time of a restore operation approaches the time to restore all memory pages (plus reading the small undo/redo logs of the disk block maps). The large jump at a restore distance of 600 seconds for PostMark is due to restoring enough data to thrash the host memory. The time for the second step depends on the distance from the checkpoint reached in step one to the target point. Since replay on TTVM occurs at approximately the same speed as the logged run, the average time of this step for a random point is half the checkpoint interval.

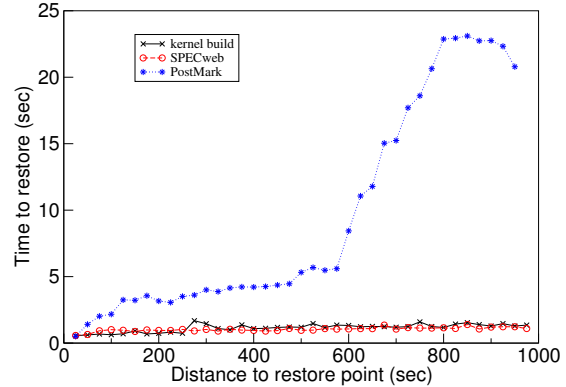


Figure 6: Time to restore to a checkpoint.

6 Experience and lessons learned

In this section, we describe our experience using TTVM to track down four kernel bugs and show how using reverse `gdb` commands simplified the process. Our experience provides anecdotal support for the intuition that reverse debugging is a useful primitive for debugging; it does not constitute an unbiased user study for quantifying the benefits of reverse debugging. After describing several anecdotes, we describe the general types of situations in which reverse debugging is most helpful and discuss the interactivity of using reverse debugging commands.

6.1 USB device driver

We first describe our experience with a non-deterministic bug that we encountered on the host OS running on our desktop computer. Our desktops use Inside Out Networks Edgeport USB serial port hubs to communicate with our test machines, but these were causing our desktop computers to crash intermittently (usually overnight). This bug provided a good test for our system. As a bug in the current host OS, it provided a realistic context for our tool. As a non-deterministic bug, it provided a chance to show the usefulness of time travel. As a bug in the host device driver, it makes use of our extensions to UML that enable host device drivers to run (and therefore be debugged) in the guest OS. Last but not least, it was getting in the way of our work.

We started by enabling in our guest OS the `io_ti` serial port hub driver and `usb-uhci` chipset driver. These drivers communicate with their devices via IN/OUT instructions, interrupts, and DMA. As expected, the drivers caused the guest OS to crash intermittently.

We first tried to debug the problem without TTVM. `gdb` showed that the crash occurred because the interrupt service routine called the kernel `schedule` function.

However, it proved difficult to deduce the sequence of events that caused the interrupt service routine to eventually call `schedule`. The call stack showed the call sites of each active function, but the number and size of these functions made it difficult to understand the sequence of events that led to the call to `schedule`. In addition, the stack contained only the current value of each variable and it was difficult to determine what had happened without seeing the prior values of each variable.

The usual approach to gain more information about the sequence of events that lead to a fault is to run the program again and step through the execution with a debugger (i.e. cyclic debugging). This approach fails for this type of bug for several reasons. First, the bug was intermittent and so would usually not appear on successive runs. Second, even if the bug did appear on a succeeding run, it would likely not appear at the same time; this makes it difficult to zero in on the bug over multiple runs. Third, bugs in device drivers pose special problems for traditional debuggers because the device may require real-time responses that cannot be met by a paused driver.

TTVM avoids these difficulties during debugging because it does not need to use the device in order to replay and debug the driver. TTVM logs all interactions with the device, including I/O, interrupts, and DMA. During replay, the driver transitions through the same sequence of states as it went through during logging (i.e. while it was driving the device), regardless of timing or the state of the device. As a result, debugging can pause the driver during replay without altering its execution.

Using TTVM, we were able to step backward through the execution of the bug and understand quickly the sequence of events that led to the call to `schedule`. Under high load, a buffer in the `tty` driver became full during an interrupt service routine invocation, and this caused the generic usb driver to call down to the `io_ti` driver, which in turn issued a configuration request to the device to throttle its communication with the computer. After issuing this configuration request, the driver waited for a response, which caused the call to `schedule`. This bug appeared in the current release of Linux 2.4 and 2.6. We also discovered a related bug which could cause the throttling routine to wait on a semaphore, and this can also cause a call to `schedule` during an interrupt service routine invocation. Using TTVM and reverse debugging, we understood the bug quickly and in enough detail to submit a patch which is being included in the Linux kernel.

6.2 System call bug

While developing TTVM, we encountered a guest kernel panic. We first tried to debug this error using traditional cyclic debugging techniques and standard `gdb`, i.e. not

using time travel. First, we set a breakpoint in the guest kernel `panic` function that is invoked when the kernel encounters an unrecoverable error. We then re-ran the virtual machine, hoping for the guest kernel panic to re-occur. Fortunately, the bug re-occurred and `gdb` gained control when a memory exception caused by guest kernel code triggered a panic. The fault occurred after the guest kernel attempted to execute an instruction at address 0. We tried to understand how the kernel reached address 0 by traversing up the call stack of the guest kernel. However, `gdb` was unable to traverse up the call stack because the most recent call frame had been corrupted when the kernel called the “function” at address 0. Since `gdb` was unable to find the prior function, we next looked at the data on the stack manually to try to find a valid return address. We found a few candidate addresses, but we eventually gave up after disassembling the guest kernel and searching through various assembly code segments.

We next used reverse commands to debug the guest kernel. We attached `gdb` to the guest-kernel host process at the time of the panic. We then performed several reverse single steps which took us to the point at which address 0 had been executed. We performed another reverse single step and found that this address had been reached from the system call handler. At this point we used a number of standard `gdb` commands to inspect the state of the virtual machine and determine the cause of the error. The bug was an incorrect entry in the system call table, which caused a function call to address 0.

6.3 Kernel race condition bug

We next tried debugging a guest kernel bug that had been posted on the UML kernel mailing list. The error we found was triggered by executing the user-mode command `ltrace strace ls`, which caused the guest kernel to panic.

First, we tried to debug the error using traditional cyclic debugging techniques and standard `gdb`, i.e. not using time travel. We set a breakpoint in the kernel `panic` function and waited for the error. After the `panic` function was called, we traversed up the call stack to learn more about how the error occurred. According to our initial detective work, the guest kernel received a debug exception while in guest kernel mode. However, debug exceptions generated during guest kernel execution get trapped by the debugger prior to delivery. Since `gdb` had not received notification of an extraneous debugging exception, we deemed a guest kernel-mode debugging exception unlikely.

By performing additional call stack traversals, we determined that the current execution path originated from a function responsible for redirecting debugging excep-

tions to guest user-mode processes. This indicated that the debugging exception occurred in guest user mode, rather than in guest kernel mode as indicated by the virtual CPU mode variable. Based on that information, we concluded that either the call stack was corrupted, or the virtual mode variable was corrupted.

We sought to track changes to the virtual mode variable in two ways, both of which failed. First, we set a forward watchpoint on the mode variable and re-ran the test. This failed because the mode variable was modified legitimately too often to examine each change. Second, we set a number of conditional breakpoints to try to narrow down the time of the corruption. With the conditional breakpoints in place, we re-ran the test case but it executed without any errors. We then gave up trying to track down this non-deterministic bug with cyclic debugging and switched to using our reverse debugging tools.

Our first step when using the reverse debugging tools was to set a reverse watchpoint on the virtual CPU mode variable. After trapping on the guest kernel panic, we were taken back to an exception handler where the variable was being changed intentionally. The new value indicated that the virtual machine was in virtual kernel mode when this exception was delivered. We reverse stepped to confirm that this was in fact the case, and then went forward to examine the subsequent execution. Since the virtual CPU mode variable is global, and the nested exception handler did not reset the value when it returned, the original exception handler (the user mode debugging exception) incorrectly determined that the debugging exception occurred while in virtual kernel mode. At this point it was clear that the exception handler should have included this variable as part of the context that is restored upon return.

It is instructive to compare our experience fixing this bug with that of a core Linux developer. Ingo Molnar was able to fix this bug in “an hour at most” by seeing the “whole state of the kernel” and because he understood the code well [20]. Ingo’s expertise apparently enabled him to deduce (from the state of the stack) the sequence of events that led to the corruption of the virtual mode variable. This approach would have been more difficult had the error manifested after the relevant stack frames had been popped. In contrast, our approach was to try to go back to the point at which the virtual mode variable was corrupted. While our naive approach failed with forward debugging, it was easy with reverse debugging and would still have worked even if the relevant stack frames had been popped.

6.4 mremap bug

Finally, we debugged a bug in the `mremap` system call (CVE CAN-2003-0985), which occurs in the

architecture-independent portion of Linux. This bug corrupts a process’s address map when the process calls `mremap` with invalid arguments; it manifests later as a kernel panic when that process exits.

First, we tried to debug the error using traditional cyclic debugging and standard `gdb`, i.e. not using time travel. We attached `gdb` when the kernel called `panic`. We traversed up the call stack and discovered that the cause of the panic was a corrupted (zero-length) address map. Unfortunately, the kernel panic occurred long after the process’s address map was corrupted, and we were unable to discern the point of the initial corruption. We thought to re-run the workload with watchpoints set on the memory locations of the variables denoting the start and end of the address map. However, these memory locations changed each run because they were allocated dynamically. Thus, while the bug crashed the system each time the program was run, the details of how the bug manifested were non-deterministic, and this prevented us from using traditional watchpoints. Even if the bug were completely deterministic, using forward watchpoints would require the programmer to step laboriously through each resulting trap during the entire run to see if the values at that trap were correct.

Reverse debugging provided a way to go easily from the kernel panic to the point at which the corruption initially occurred. After attaching `gdb` at the kernel panic, we set a reverse watchpoint on the memory locations of the variables denoting the start and end of the address map. This watchpoint took us to when the guest OS was executing the `mremap` that had been passed invalid arguments, and at this point it was obvious that `mremap` had failed to validate its arguments properly.

6.5 Lessons learned

We learned four main lessons from our experience about the types of bugs that TTVM helps debug.

First, we learned that many bugs are too fragile to find with cyclic debugging. Heisenbugs [13] such as race conditions thwart cyclic debugging because they manifest only occasionally, depending on transient conditions such as timing. However, cyclic debugging often fails even for bugs that manifest each time a program runs, because the details of how they manifest change from run to run. Details like the internal state of an OS depend on numerous, hard-to-control variables, such as the sequence and scheduling order of all processes that have been run since boot. In the case of the `mremap` bug, minor changes to the internal OS state (the address of dynamically allocated kernel memory) prevented us from using watchpoints during cyclic debugging.

In contrast, TTVM’s reverse debugging makes even the most fragile of bugs perfectly repeatable. TTVM’s

deterministic replay ensures that the details of the internal OS state will remain consistent from run to run and thus enables the use of debugging commands that depend on fragile information.

Second, we learned that bugs that take a long time to trigger highlight the poor match between standard debugging commands and most debugging scenarios. Standard watchpoints and breakpoints are best suited to go to a *future* point of possible interest. In contrast, programmers usually want to go to a *prior* point of possible interest, because they are following in reverse the chain of events between the execution of the buggy code and the ensuing detection of that error. Trying to go backward by re-running the workload with forward watchpoints and breakpoints is very clumsy without TTVM. If the bug is fragile, the bug may not manifest (or may not manifest in the same way) during each run. Even if the bug manifests in exactly the same way during each run, cyclic debugging forces a programmer to step manually through all spurious traps since the beginning of the run, or to run the program numerous times searching manually for the period of interest.

In contrast, TTVM's reverse debugging commands provided exactly the semantics we needed to find each of the bugs we encountered. For the kernel race bug and the mremap bug, the point of interest was the last time a variable changed before the error was detected. For the system call bug, the point of interest was a few instructions before the error was detected.

Third, we learned that standard debuggers are difficult to use for bugs that corrupt the stack or that are detected after the relevant stack frame is popped. Standard debuggers approximate time travel by traversing up the call stack, but this form of time travel is neither complete nor reliable. Stack traversal is incomplete because it shows only the values of variables on the stack and because it shows those variables only at the time of their function's last invocation. For the mremap bug, the code that contained the error was executed during a prior system call and was not on the stack when the error was detected. Stack traversal is unreliable because it works only if the stack is intact. For the system call bug, the stack had been corrupted by an erroneous function call. Similarly, common buffer overflow attacks corrupt the stack and render stack traversal difficult. It is ironic that one of the most powerful techniques of standard debuggers depends on the partial correctness of the program being debugged.

In contrast, TTVM provides complete and reliable time travel. It is complete in that it can show the state of any variable at any time in the past. It is reliable in that it works without depending on the correctness of the program being debugged.

Finally, we learned that bugs in device drivers are particularly hard to solve with cyclic debugging. Device

driver bugs are often non-deterministic because they depend on interrupts and device inputs. In addition, devices may require real-time responses that cannot be met by a paused driver. In contrast, TTVM allows one to replay device drivers deterministically, and TTVM's replay works without interacting with the device.

6.6 Interactivity of reverse debugging

To debug the bugs described in this section we triggered the error while logging, then replayed the virtual machine to diagnose the error. When replaying, we set the checkpoint interval to ten seconds. This checkpoint interval added reasonable runtime overhead for debugging (in fact, it added less overhead than some forward debugging commands, such as conditional breakpoints) and was short enough to support interactive performance for reverse commands.

We found the reverse commands to be quite interactive. Usually we used the reverse commands to step back a couple instructions or to go back to a recent breakpoint within the current checkpoint interval. This caused most of our checkpoint state to remain in the host file cache, which further sped up subsequent reverse commands. Restoring to the nearest checkpoint took under 1 second; replaying to the point of interest took five seconds on average (given the ten second checkpoint interval). Taking a reverse single step took about 12 seconds on average; this includes the time for both passes (Figure 3), i.e. restoring the checkpoint twice and replaying the remainder of the checkpoint interval twice. Overall, we found the speed of our reverse debugging commands fast enough to support interactive usage comfortably.

7 Related work

Our work draws on techniques from several areas, including prior work on reverse execution of deterministic programs, replay of non-deterministic programs, and virtual-machine replay. Our unique contribution is combining these techniques in a way that enables powerful debugging capabilities that have not been available previously for systems (such as operating systems) that have numerous sources of non-determinism, that run for long periods of time, or that interact with hardware devices.

Re-executing prior computation through checkpoint and logging has been discussed in the programming community for many years [30, 10, 1, 4, 6, 24]. However, no prior reverse debugger would work for operating systems or for a user-level virtual machine such as User-Mode Linux. The primary limitation of prior systems is they are unable to replay programs with non-deterministic effects such as interrupts and thread scheduling [10, 4, 1,

6]. User-Mode Linux simulates interrupts and preemptions with asynchronous signals, and prior reverse debuggers are not able to replay such events. In addition, most reverse debuggers implement time travel by logging all changes to variables [30, 1, 21, 6], and this approach logs too much data when debugging long-running systems such as an OS. Finally, some systems work at the language level [27], and this prevents them from working with operating systems in a different language or with application binaries.

Researchers have worked to replay non-deterministic programs through various approaches. The events of different threads can be replayed at different levels, including logging accesses to shared objects [16], logging the scheduling order of multi-threaded programs on a uniprocessor [22], or logging physical memory accesses in hardware [2]. Other researchers have worked to optimize the amount of data logged [21].

Virtual-machine replay has been used for non-debugging purposes. Hypervisor used virtual-machine replay to synchronize the state of a backup machine to provide fault tolerance [5]. ReVirt used virtual-machine replay to enable detailed intrusion analysis [9]. Our work applies virtual-machine replay to achieve a new capability, which is reverse debugging of operating systems. TTVM also supports additional features over prior virtual-machine replay systems. TTVM supports the ability to run, log, and replay real device drivers in the guest OS, whereas prior virtual-machine replay systems ran only para-virtualized device drivers in the guest OS. In addition, TTVM can travel quickly forward and backward in time through its use of checkpoints and undo and redo logs, whereas ReVirt supported only a single checkpoint of a powered-off virtual machine and Hypervisor did not need to support time travel at all (it only supported replay within an epoch).

Another approach for providing time travel is to use a complete machine simulator, such as Simics [18]. Simics supports deterministic replay for operating systems and applications and has an interface to a debugger. However, Simics is drastically slower than TTVM, and this makes debugging long runs impractical. On a 750 MHz Ultrasparc III, Simics executes 2-6 million x86 instructions per second (several hundred times slower than native) [18], whereas virtual machines typically incur a slowdown of less than 2x.

8 Conclusions and future work

We have described the design and implementation of a time-traveling virtual machine and shown how to use TTVM to add powerful capabilities for debugging operating systems. We integrated TTVM with a general-

purpose debugger, implementing commands such as reverse breakpoint, reverse watchpoint, and reverse step.

TTVM added reasonable overhead in the context of debugging. The logging needed to support time travel for three OS-intensive workloads added 3-12% in running time and 2-85 KB/sec in log space. Taking checkpoints every minute added less than 4% time overhead and 1-5 MB/sec space overhead. Taking checkpoints every 10 second to prepare for debugging a portion of a run added 16-33% overhead and enabled reverse debugging commands to complete in about 12 seconds.

We used TTVM and our new reverse debugging commands to fix four OS bugs that were difficult to find with standard debugging tools. We found the reverse debugging commands to be intuitive to understand and fast and easy to use. Reverse debugging proved especially helpful in finding bugs that were fragile due to non-determinism, bugs in device drivers, bugs that required long runs to trigger, bugs that corrupted the stack, and bugs that were detected after the relevant stack frame was popped.

Possible future work includes exploring non-traditional debugging operations that are enabled by time travel and deterministic replay. For example, one could measure the effects of a programmer-induced change by forking the execution and comparing the results after the change with the results of the original run.

9 Acknowledgments

Our shepherd, Steve Gribble, and the anonymous reviews provided feedback that helped improve this paper. This research was supported in part by ARDA grant NBCHC030104, National Science Foundation grants CCR-0098229 and CCR-0219085, and by Intel Corporation. Samuel King was supported in part by a National Defense Science and Engineering Graduate Fellowship.

References

- [1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An Execution-Backtracking Approach to Debugging. *IEEE Software*, 8(3), May 1991.
- [2] D. F. Bacon and S. C. Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991.
- [3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.
- [4] B. Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 299-310, June 2000.

- [5] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault-Tolerance. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 1–11, December 1995.
- [6] S.-K. Chen, W. K. Fuchs, and J.-Y. Chung. Reversible Debugging Using Program Instrumentation. *IEEE Transactions on Software Engineering*, 27(8):715–727, August 2001.
- [7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proceedings of the 2001 Symposium on Operating Systems Principles*, pages 73–88, October 2001.
- [8] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, October 2000.
- [9] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, December 2002.
- [10] S. I. Feldman and C. B. Brown. IGOR: a system for program debugging via reversible execution. In *Proceedings of the 1988 ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123, November 1988.
- [11] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge Computer Laboratory, August 2004.
- [12] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34–45, June 1974.
- [13] J. Gray. Why do computers stop and what can be done about it? In *Proceedings of the 1986 Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, January 1986.
- [14] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR3022, Network Appliance, October 1997.
- [15] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Technical Conference*, pages 71–84, June 2003.
- [16] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, pages 471–482, April 1987.
- [17] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.
- [18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.
- [19] J. M. Mellor-Crummey and T. J. LeBlanc. A Software Instruction Counter. In *Proceedings of the 1989 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78–86, April 1989.
- [20] I. Molnar, February 2005. personal communication.
- [21] R. H. B. Netzer and M. H. Weaver. Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs. In *Proceedings of the 1994 Conference on Programming Language Design and Implementation*, June 1994.
- [22] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the 1996 Conference on Programming Language Design and Implementation*, pages 258–266, May 1996.
- [23] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, December 2002.
- [24] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A light-weight rollback and deterministic replay extension for software debugging. In *Proceedings of the 2004 USENIX Technical Conference*, June 2004.
- [25] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.
- [26] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Technical Conference*, June 2001.
- [27] A. Tolmach and A. W. Appel. A Debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.
- [28] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.
- [29] M. Xu, R. Bodik, and M. D. Hill. A “Flight Data Recorder” for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 2003 International Symposium on Computer Architecture*, June 2003.
- [30] M. V. Zelkowitz. Reversible execution. *Communications of the ACM*, 16(9):566, September 1973.

Using Valgrind to detect undefined value errors with bit-precision

Julian Seward
OpenWorks LLP
Cambridge, UK
julian@open-works.co.uk

Nicholas Nethercote
Department of Computer Sciences
University of Texas at Austin
njn@cs.utexas.edu

Abstract

We present Memcheck, a tool that has been implemented with the dynamic binary instrumentation framework Valgrind. Memcheck detects a wide range of memory errors in programs as they run. This paper focuses on one kind of error that Memcheck detects: undefined value errors. Such errors are common, and often cause bugs that are hard to find in programs written in languages such as C, C++ and Fortran. Memcheck's definedness checking improves on that of previous tools by being accurate to the level of individual bits. This accuracy gives Memcheck a low false positive and false negative rate.

The definedness checking involves shadowing every bit of data in registers and memory with a second bit that indicates if the bit has a defined value. Every value-creating operation is instrumented with a shadow operation that propagates shadow bits appropriately. Memcheck uses these shadow bits to detect uses of undefined values that could adversely affect a program's behaviour.

Under Memcheck, programs typically run 20–30 times slower than normal. This is fast enough to use with large programs. Memcheck finds many errors in real programs, and has been used during the past two years by thousands of programmers on a wide range of systems, including OpenOffice, Mozilla, Opera, KDE, GNOME, MySQL, Perl, Samba, The GIMP, and Unreal Tournament.

1 Introduction

The accidental use of undefined values is a notorious source of bugs in programs written in imperative languages such as C, C++ and Fortran. Such *undefined value errors* are easy to make, but can be extremely difficult to track down manually, sometimes lurking unfound for years.

In this paper we describe Memcheck, a practical tool which detects a wide range of memory errors, including undefined value errors. Memcheck is implemented using the dynamic binary instrumentation framework Valgrind [10, 9].

1.1 Basic operation and features

Memcheck is a dynamic analysis tool, and so checks programs for errors as they run. Memcheck performs four kinds of memory error checking.

First, it tracks the addressability of every byte of memory, updating the information as memory is allocated and freed. With this information, it can detect all accesses to unaddressable memory.

Second, it tracks all heap blocks allocated with `malloc()`, `new` and `new[]`. With this information it can detect bad or repeated frees of heap blocks, and can detect memory leaks at program termination.

Third, it checks that memory blocks supplied as arguments to functions like `strcpy()` and `memcpy()` do not overlap. This does not require any additional state to be tracked.

Fourth, it performs *definedness checking*: it tracks the definedness of every bit of data in registers and memory. With this information it can detect undefined value errors with bit-precision.

All four kinds of checking are useful. However, of the four, definedness checking is easily the most sophisticated, and it is this checking that this paper focuses on.

Memcheck uses dynamic binary instrumentation to instrument the program to be checked (the *client*) on-the-fly at run-time. Execution of the added instrumentation code is interleaved with the program's normal execution, not disturbing normal program behaviour (other than slowing it down), but doing extra work “on the side” to detect memory errors. Because it instruments and analyses executable machine code, rather than source code or object code, it has the following nice properties.

- Wide applicability: it works with programs written in any language.¹
- Total coverage: all parts of the client are executed under Memcheck's control, including dynamically linked libraries and the dynamic linker, even if the

```

int main(void) {
    int x, y, z, *p;
    char buf[10];
    write(1, buf, 1);           // bug 1
    x = ( x == 0 ? y : z );     // bug 2
    return *p + x;              // bug 3
}

```

Figure 1: Example program `badprog.c`

source code is not available on the system. Indeed, it is only by doing this that Memcheck can be accurate—partial coverage leads either to lots of missed errors (false negatives) or lots of invalid errors (false positives).

- Ease of use: unlike many similar tools, it does not require programs to be prepared (e.g. recompiled or relinked) in any way.

Memcheck is part of the Valgrind suite, which is free (GPL) software, and is available for download from the Valgrind website [14]. It currently runs only on the x86/Linux platform, although work is currently underway to port it to other platforms such as AMD64/Linux, PowerPC/Linux, x86/FreeBSD, and PowerPC/MacOSX.

1.2 Using Memcheck

Memcheck is easy to use. As an example, consider the (contrived) program `badprog.c` in Figure 1. It contains three undefined value errors. To check the compiled program `badprog` the user only has to type:

```
valgrind --tool=memcheck badprog
```

The `--tool=` option specifies which tool in the Valgrind suite is used. The program runs under Memcheck’s control, typically 20–30 times slower than usual. This slow-down is partly due to Memcheck’s definedness checking, partly due to its other checking, and partly due to Valgrind’s inherent overhead. The program’s output is augmented by Memcheck’s output, which goes by default to standard error, although it can be redirected to a file, file descriptor, or socket with a command line option.

Figure 2 shows the resulting output. The first three lines are printed by Memcheck on startup. The middle section shows three error messages issued by Memcheck. The final three lines are printed at termination, and summarise Memcheck’s findings. Each line of Memcheck’s output is prefixed with the client’s process ID, 27607 in this case.

The three error messages are quite precise. The first indicates that the memory passed to the system call `write()` via the `buf` argument contains undefined values; its last line indicates that the undefined value is on

```

void set_bit ( int* arr, int n ) {
    arr[n/32] |= (1 << (n%32));
}

int get_bit ( int* arr, int n ) {
    return 1 & (arr[n/32] >> (n%32));
}

int main ( void ) {
    int* arr = malloc(10 * sizeof(int));
    set_bit(arr, 177);
    printf("%d\n", get_bit(arr, 178));
    return 0;
}

```

Figure 3: Unsafe use of a bit-array

the stack of thread 1 (the main thread). The second indicates that a conditional jump depends on an undefined value. The third indicates that an undefined value is used as a pointer. All three error messages include a stack trace that indicate precisely where the error is occurring. In order to get exact line numbers in the stack traces, the client must be compiled with debugging information. If this is missing, the code locations are less precise, as can be seen with the location within the `write()` function (not to be confused with the `write()` system call)—GNU libc on this system was not compiled with debugging information.

Attentive readers may note that the final line of `badprog.c` could cause a segmentation fault due to the use of the uninitialised variable `p` as an address. On one system we tried this test on, exactly that happened, and so Memcheck issued an additional “Invalid read of size 4” warning immediately before the memory access, thanks to its addressability checking. For the run presented, the memory access (un)luckily hit addressable memory, so no addressability error message was issued.

1.3 Bit-level precision

Memcheck is the first tool we are aware of that tracks definedness down to the level of bits. Other tools track definedness at byte granularity (Purify) or word granularity (Third Degree).

This means Memcheck correctly handles code which deals with partially-defined bytes. In C and C++, two common idioms give rise to such bytes: use of bit arrays and use of bitfields in structures. Tools which track definedness at byte or word granularities necessarily give inaccurate results in such situations – either they fail to report genuine errors resulting from uses of uninitialised bits, or they falsely flag errors resulting from correct uses of partially defined bytes.

The program shown in Figure 3 uses an uninitialised bit in a bit-array. Memcheck reports this, but Purify does not². Memcheck is known to have found previously-

```

==27607== Memcheck, a memory error detector for x86-linux.
==27607== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward et al.
==27607== For more details, rerun with: -v
==27607==
==27607== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==27607==   at 0x420D2473: write (in /lib/tls/libc-2.3.2.so)
==27607==   by 0x8048347: main (badprog.c:6)
==27607== Address 0x52BFE880 is on thread 1's stack
<junk character printed by write()>
==27607==
==27607== Conditional jump or move depends on uninitialised value(s)
==27607==   at 0x804834F: main (badprog.c:7)
==27607==
==27607== Use of uninitialised value of size 4
==27607==   at 0x804836B: main (badprog.c:8)
==27607==
==27607== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 11 from 1)
==27607== malloc/free: in use at exit: 0 bytes in 0 blocks.
==27607== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.

```

Figure 2: Example output for `badprog.c`

undetected uses of single uninitialised bits in C++ structure bitfields, in at least one large, widely used C++ code base.

1.4 Contributions

Our main contribution is a detailed description of Memcheck's definedness checking, which has only been briefly touched upon in previous publications about Valgrind [10, 9]. Memcheck's definedness checking improves on that performed by previous tools by being accurate to the level of individual bits. Its false positive and false negative rates are very low. Finally, the run-time overhead of the definedness checking is reasonable enough that it is practical to use on very large programs.

1.5 Paper structure

The rest of this paper is structured as follows. Section 2 describes how Memcheck works, in particular the details of shadow bit operations, which are crucial in ensuring Memcheck's accuracy and speed. Section 3 evaluates Memcheck by considering the cost of its use—in terms of how easy it is to obtain and run, the ease of using its results, and its impact on performance—and the benefits provided by its bug-finding abilities. Section 4 discusses related work. Section 5 concludes.

2 How Memcheck works

2.1 Valgrind

Memcheck is implemented as a plug-in to Valgrind [10, 9]. Valgrind is a framework for creating tools that use dy-

namc binary instrumentation; it does all the hard work of inserting instrumentation into machine code at run-time. Tools are created as plug-ins, written in C, to Valgrind's *core*. The basic view is:

Valgrind core + tool plug-in = Valgrind tool.

The resulting tool loads the client at start-up, grafting itself onto the process as it does so. It then starts executing the client, by (re)compiling the client's code, one basic block at a time, in a just-in-time, execution-driven fashion. The compilation process involves disassembling the machine code into an intermediate representation called UCode. UCode is instrumented by the tool plug-in, and the instrumented UCode is then converted back into x86 code. The resulting code is stored in Valgrind's code cache, to be rerun as necessary.

The core spends most of its execution time making, finding, and running translations. None of the client's original code is run. The core also provides many services to tools, to ease common tasks such as recording errors and reading debug information. Only one tool can be used at a time.

The Valgrind distribution contains the core, plus five tools: Memcheck, Addrcheck (a lightweight version of Memcheck that omits definedness checking), a cache profiler, a memory space-use (heap) profiler, and a data race detector for POSIX pthread-ed programs.

2.2 Overview

The basic idea underlying the definedness checking is straightforward.

- Every single bit of data, *b*, maintained by a program, in both registers and memory, is *shadowed*

by a piece of metadata, called a *definedness* bit. For historical reasons these are often also referred to as *V bits* (V being short for “validity”). Each V bit indicates whether or not the bit *b* it shadows is regarded as currently having a properly defined value.

- Every single operation that creates a value is shadowed by a *shadow operation* that computes the V bits of any outputs, based on the V bits of all inputs and the operation. The exact operations performed by this *shadow computation* are important, as they must be sufficiently fast to be practical, and sufficiently accurate to not cause many false positives.
- Every operation that uses a value in such a way that it could affect the observable behaviour of a program is checked. If the V bits indicate that any of the operation’s inputs are undefined, an error message is issued. The V bits are used to detect if any of the following depend on undefined values: control flow transfers, conditional moves, addresses used in memory accesses, and data passed to system calls.

Most operations do not directly affect the program’s observable behaviour. In such cases, the V bits are not checked. Instead, V bits for the result of the operation are computed. Hence, for the most part, Memcheck silently tracks the flow of undefined values, and only issues an error message when use of such a value is potentially dangerous. This scheme is required because undefined values are often copied around without any problem, due to the common practice, used by both programmers and compilers, of padding structures to ensure fields are word-aligned.

From this overview, the main overheads of definedness checking are apparent. First, the use of V bits doubles the amount of memory in use. Second, most instructions compute new values, and thus require a shadow operation itself consisting of one or more instructions.

2.3 Details of V bits

A V bit of zero indicates that the corresponding data bit has a properly defined value, and a V bit of one indicates that it does not. This is counterintuitive, but makes some of the shadow operations more efficient than they would be if the bit values were inverted.

Every 32-bit general purpose register is shadowed by a 32-bit shadow register, and every byte of memory has a shadow V byte. After that, there are some exceptions to the basic rule of “one V bit per data bit”.

- The x86 condition code register (%eflags) is approximated with a single V bit, since tracking all

6 condition codes individually is expensive and mostly unnecessary.

- The program counter is not shadowed. Instead, we regard it as always defined, and emit an error message whenever a conditional jump depends on undefined condition codes. One way to interpret such a jump is as an attempt to assign an undefined value to the program counter, but since we are not tracking the program counter’s definedness state, we regard it as always-defined, and so must immediately report any attempt to “assign” it an undefined value.
- Floating point, MMX and SSE registers are not shadowed. Memcheck can still perform definedness checking on code using these registers, but such checks may produce a higher false positive rate than would have occurred had the registers been shadowed.

2.4 Instrumentation basics

In principle it is possible to directly state the V bit transformations required to shadow each x86 instruction. In practice, the x86 instruction set is so complex and irregular that this would be difficult and fragile. Fortunately, UCode (Valgrind’s intermediate representation) is RISC-like and clearly exposes all memory and arithmetic operations, which makes Memcheck’s instrumentation task much easier.

Another important aspect of UCode is that it supports the use of an infinite supply of virtual registers. The initial translation from x86 is expressed in terms of such registers. Memcheck interleaves its own instrumentation UCode with it, using as many new virtual registers as required. Valgrind’s core has the job of performing instruction selection and register allocation to convert this sequence back to executable x86 code.

2.5 Abstract operations on V bits

The V bit instrumentation scheme is best described in terms of a family of simple abstract operations on V bits. We will use *d1* and *d2* to denote virtual registers holding real values, and *v1* and *v2* denote virtual shadow registers holding V bits. Also, some operations below use *X* and *Y* indicate the operand and result widths in bytes (and 0 represents an operand with a width of a single bit). Those operations for which the width is not specified are width-independent.

Memcheck uses the following binary operations.

- *DifD(v1, v2)* (“defined if either defined”) returns a V bit vector the same width as *v1* and *v2*.

Each bit of the result indicates definedness if either corresponding operand bit indicates definedness. Since our encoding is zero for defined and one for undefined, `DifD` can be implemented using a bitwise-AND operation.

- `UifU(v1, v2)` (“undefined if either undefined”) dually propagates undefinedness from either operand, again at a bitwise level, and can be implemented using a bitwise-OR operation.
- `ImproveAND(d, v)` and `ImproveOR(d, v)` are helpers for dealing with AND and OR operations. These have the interesting property that the resulting `V` bits depend not only on the operand `V` bits, but also on the operand data values. `ImproveAND` takes a data value (`d`) and a `V` bit value (`v`), and produces an “improvement value” bitwise as follows:

```
ImproveAND(0, undefined) = undefined
ImproveAND(0, defined)   = defined
ImproveAND(1, undefined) = undefined
ImproveAND(1, defined)   = undefined
```

The second case is the interesting one. If one of the arguments is a defined zero bit, we don’t care that the other argument might be undefined, since the result will be zero anyway. Hence `ImproveAND` creates a “defined” `V`-bit, which, as described in Section 2.6, is merged into the final result for AND using `DifD`. This has the effect of forcing that bit of the result to be regarded as defined. “undefined” is the identity value for `DifD`, so the other three cases have no effect on the final outcome.

For exactly analogous reasons, `ImproveOR` behaves similarly, with the interesting case being `ImproveOR(1, defined) = defined`.

The following unary operations are also needed.

- `Left(v)` simulates the worst-case propagation of undefinedness upwards (leftwards) through a carry chain during integer add or subtract. `Left(v)` is the same as `v`, except that all bits to the left of the rightmost 1-bit in `v` are set. For example, using 8-bit values, `Left(00010100) = 11111100`. `Left` can be implemented in two x86 instructions, a negation followed by an OR operation.
- `PCastXY(v)` are a family of size-changing operations which can be interpreted as “pessimising casts”. If *all* bits of the operand are zero (defined), `PCast` produces a `V` bit vector at the new width in which all bits are zero (defined),

else it produces a value with all bits one (undefined). For example, `PCast12(00010100) = 1111111111111111`, and `PCast12(00000000) = 0000000000000000`.

These casts are used in various approximations in which the definedness checking needs to consider the worst-case across a whole word of bits. It is important to appreciate that the narrowing casts (where $X > Y$) do not simply discard the high bits of the operand. Similarly, the case where $X = Y$ is not the identity function. In all cases, each result bit depends on all operand bits, regardless of their relative sizes. `PCast` can be implemented in at most three x86 instructions: for narrowing casts, negation followed by subtract-with-borrow; for widening casts, a shift, a negation, and a subtract-with-borrow.

- `ZWidenXY(v)` are a family of widening operations which mimic unsigned (zero-extend) widening of data values. As with `PCast`, `X` and `Y` denote argument and result widths, with the additional requirement that $Y \geq X$. Zero-widening a data value produces zeroes in the new positions, and so `ZWiden` needs to indicate these new positions are defined. Since defined values are encoded as zero bits, `ZWiden` can itself be implemented using a zero-widen instruction. This is the first of several cases where choosing zero (rather than one) to mean “defined” simplifies the implementation.
- `SWidenXY(v)` is the dual family of signed widening operations. A signed widening copies the top argument bit into all new positions. Therefore `SWiden` has to copy the top definedness bit into all new positions and so can itself be implemented using a signed-widen instruction.

2.6 The instrumentation scheme proper

Every operation (instruction or system call) that creates a value must be instrumented with a shadow operation that computes the corresponding `V` bits. This section describes these shadow operations in detail.

Recall that for each virtual register `d1, d2...` in the incoming `UCode`, `Memcheck` allocates a shadow virtual register `v1, v2...` to carry the corresponding `V` bits.

Register and memory initialisation At startup, all registers have their `V` bits set to one, i.e. undefined. The exception is the stack pointer, which has its `V` bits set to zero, i.e. defined.

All memory bytes mapped at startup (i.e. code and data segments, and any shared objects) have their `V` bits set to zero (defined).

Memory allocation and deallocation The Valgrind framework intercepts function and system calls which cause usable address ranges to appear/disappear. Memcheck is notified of such events and marks shadow memory appropriately. For example, malloc and mmap bring new addresses into play: mmap makes memory addressable and defined, whilst malloc makes memory addressable but undefined. Similarly, whenever the stack grows, the newly exposed area is marked as addressable but undefined. Whenever memory is deallocated, the deallocated area also has its values all marked as undefined.

Memcheck also uses such events to update its maps of which address ranges are legitimately addressable. By doing that it can detect accesses to invalid addresses, and so report to the user problems such as buffer overruns, use of freed memory, and accesses below the stack pointer. Details of this addressability checking are beyond the scope of this paper.

Literals All literals are, not surprisingly, considered to be completely defined.

Data copying operations These are straightforward. Register-to-register moves give rise to a move between the corresponding shadow registers. Register-to-memory and memory-to-register transfers are instrumented with a move between the corresponding shadow register and shadow memory.

x86 contains a byte-swap (endianness-change) instruction. As this merely rearranges bits in a word, a byte-swap instruction is also applied to the shadow value.

Addition and subtraction Given $d3 = \text{Add}(d1, d2)$ or $d3 = \text{Sub}(d1, d2)$, each result bit can simplistically be considered defined if both the corresponding argument bits are defined. However, a result bit could also be undefined due to an undefined carry/borrow propagating upwards from less significant bit positions. Therefore Memcheck needs to generate $v3 = \text{Left}(\text{UifU}(v1, v2))$.

The same scheme is used for multiplies. This is overly conservative because the product of two numbers with N and M consecutive least-significant defined bits has $N + M$ least-significant defined bits, rather than $\min(N, M)$ as the Add/Sub scheme generates. It would be possible to do a better job here, but the extra expense does not seem justified given that very few, if any, complaints have arisen over the subject of false positives arising from multiplies.

The shadow operation for Neg (negation) is trivially derived by constant-folding the shadow operation for $\text{Sub}(0, d)$, giving the result $\text{Left}(v)$, where v is the shadow for d .

Add with carry and subtract with borrow These take the CPU's carry flag as an additional single-bit operand. Let vfl be the virtual 1-bit-wide register tracking the definedness of the condition codes (%eflags). If this extra operand is undefined, the entire result is undefined, so the following formulation derives straightforwardly from the Add/Sub case:

```
v3 = UifU( Left(UifU(v1, v2)),
           PCast0X(vfl) )
```

where X is the width of $v1, v2$ and $v3$.

Xor A simple case: given $d3 = \text{Xor}(d1, d2)$, generate $v3 = \text{UifU}(v1, v2)$.

The rule for Not is trivially derived by constant-folding the rule for $\text{Xor}(0xFF \dots FF, d)$, giving, as one might expect, the simple result v , where v is the shadow for d ; i.e. the V bits are unchanged.

And and Or These require inspection of the actual operand values as well as their shadow bits. We start off with a bitwise UifU of the operands, but fold in, using DifD, "improvements" contributed by defined zero-arguments (for And) or defined one-arguments (for Or). So, given:

```
d3 = And(d1, d2)
d3 = Or(d1, d2)
```

the resulting instrumentation assignments are, respectively:

```
v3 = DifD( UifU(v1, v2),
           DifD( ImproveAND(d1, v1),
                 ImproveAND(d2, v2) ) )
v3 = DifD( UifU(v1, v2),
           DifD( ImproveOR(d1, v1),
                 ImproveOR(d2, v2) ) )
```

This means instrumentation of And/Or is quite expensive. However, such instructions are often used with one constant operand, in which case Memcheck's post-instrumentation cleanup pass can fold these expressions down to a single ImproveAND/OR term.

Shl, Shr, Sar, Rol, Ror (Shift left, Unsigned shift right, Signed shift right, Rotate left, Rotate right). In all cases, if the shift/rotate amount is undefined, the entire result is undefined. Otherwise, for reasons which are somewhat subtle, the result V bits are obtained by applying the same shift/rotate operation to the V bits of the value to be shifted/rotated.

Given input $d3 = \text{OP}(d1, d2)$, where $d2$ is the shift/rotate amount, and the sizes of $d1/d3$ and $d2$ are respectively X and Y , the resulting instrumentation assignment is

```
v3 = UifU( PCastYX(v2), OP(v1,d2) )
```

In all five cases, the definedness bits are processed using the same operation as the original. For `ROL` and `ROR`, the definedness bits must be rotated exactly as the data bits are. `SHL` and `SHR` shift zeroes into the data, and so corresponding zeroes—indicating definedness—need to be shifted into the definedness word. `SAR` copies the top bit of the data, and so needs to also copy the top bit of the definedness word.

Widening and narrowing conversions A narrowing conversion on data throws away some of the top bits of the word, and so the same operation can be used to throw away the top bits of the shadow word.

Signed and unsigned widening conversions give rise respectively to a single `SWiden` or `ZWiden` operation.

Instructions which set flags On x86, most integer arithmetic instructions set the condition codes (`%eflags`) and Memcheck duly tracks the definedness state of `%eflags` using a single shadow bit. When an integer operation sets condition codes, it is first instrumented as described above. Memcheck pessimistically narrows the result value(s) of the shadow operation using `PCastX0` to derive a value for the `%eflags` shadow bit.

Loads, stores, conditional branches and conditional moves These are discussed in the next section.

Floating point (FP) and MMX, SSE, SSE2 (SIMD) operations Valgrind does not disassemble floating point or SIMD instructions to the same level of detail as it does integer instructions. Instead, it merely modifies some of the register fields in the instruction, marks any instructions referencing memory as such, and copies them otherwise unchanged into the output instruction stream.

Because of this, Memcheck can only offer crude instrumentation of such instructions. Such instrumentation is safe in the sense that all uses of undefined values, and all illegitimate memory accesses, will still be caught. The crudeness of the instrumentation has the effect that some computations, when done with FP or SIMD registers, may elicit false-positive undefined value errors, when similar or identical operations done using integer registers would not. The most notable case is that copying undefined data through the FP or SIMD registers will elicit false positives.

The instrumentation scheme is as follows. Neither the FP nor SIMD registers have any associated V bits. When a value is loaded from memory into such a register, if any part of the value is undefined, an error message is issued. When a value is written from such a register to memory,

shadow memory is marked as defined. This is in keeping with the **Eager** approximation scheme described shortly.

So far, this crude scheme has proven adequate, mostly because programmers and compilers rarely copy and manipulate partially-undefined data through FP or SIMD registers. However, vectorising compilers for SIMD architectures are becoming increasingly common [8], and this scheme cannot continue much longer—it is the biggest weakness of Memcheck. A new version of Memcheck under development will shadow data in floating point registers and in individual lanes of SIMD registers, thus remedying this deficiency.

Approximating everything else The above cases give sufficient accuracy to achieve a near-zero false positive rate on almost all compiler-generated and handwritten code. There are a multitude of other cases which *could* be tracked accurately, but for which there appears to be no point. These include: division, rotates through the carry flag, and calls to helper functions which implement obscure features (CPUID, RDTSC, BCD arithmetic, etc).

In such situations, two approximation schemes are possible.

- **Lazy.** The V bits of all inputs to the operation are pessimistically summarised into a single bit, using chains of `UifU` and/or `PCastX0` operations. The resulting bit will indicate “undefined” if any part of any input is undefined. This bit is duplicated (using `PCast0X`) so as to give suitable shadow output word(s) for the operation.

Using this scheme, undefinedness can be made to “flow” through unknown operations, albeit in a pessimistic manner. No error messages will be issued when such operations execute.

- **Eager.** As with **Lazy**, a summary definedness bit is pessimistically computed. If the bit is one (undefined), an error message is issued. Regardless of the bit’s value, shadow output word(s) are created indicating “defined”. Counterintuitive as this may seem, it stops cascades of undefined value error messages being issued; only the first observation of such values are reported.

Memcheck currently uses **Eager** for all floating point and SIMD operations, as described above, and **Lazy** in all other situations.

2.7 Deciding when to issue error messages

At every point where an undefined value could be consumed by an operation, Memcheck has a choice: should

it report the error right now, or should it silently propagate the undefinedness into the result? Both approaches have advantages.

- Reporting the error sooner (the eager strategy mentioned above) makes it easier for users to track down the root cause of undefined values. Undefined value errors originate primarily from reading uninitialised memory. Such values propagate through the computation until they hit a check point. If check points are rare, that path can be long, and users may have to trace back through multiple levels of procedure calls and through their data structures to find the root cause.
- Deferring error checking and reporting has two major advantages. Firstly, error checks are expensive—a test and conditional jump—and so minimising them improves performance. Secondly and more importantly, reporting errors too soon can lead to false positives: undefined values might be used in a safe way and then discarded, so an early check on them would give a pointless error to the user.

Memcheck mostly takes the second alternative, deferring error reporting as far as it can. Checks for undefined values are made only when the program is in immediate danger of performing one of the following actions, which could change its observable behaviour.

- Taking a memory exception due to use of an undefined address in a load or store.³
- Making a conditional jump based on undefined condition codes.
- Passing undefined values to a system call.
- Loading uninitialised values from memory into a SIMD or FP register.

Accordingly, instrumentation for such events is as follows.

- Memory access (all kinds): check address for definedness and issue an error message if any address bit is undefined.
- Conditional jump: check the V bit which shadows %eflags, and issue an error message if undefined.
- System call: check arguments (scalar and in memory) to the extent possible, and issue an error message if undefined values are being passed to the kernel. This requires in-depth knowledge of the kernel interface.

- Memory load into a SIMD or FP register: in addition to checking definedness of the address, also check the loaded data for definedness, and issue an error message if necessary.

Conditional moves could be handled using either the eager or lazy scheme. Memcheck handles them eagerly, testing the condition code and reporting any error immediately.

Each error check consists of testing a shadow virtual register⁴ against zero for any undefinedness, and calling a helper function to issue an error message if so. An important but non-obvious extra step is that, immediately following the test, the shadow register should be set to zero (“all defined”). Doing so prevents subsequent checks on it issuing essentially duplicate errors, which would confuse users. Consider the following C fragment:

```
int* p; /* not defined */
... = *p;
*p = ...
```

For the load, p’s shadow is tested, and an error message is issued if necessary. Subsequently in the store, reporting another such error for p would not help lead users to the root cause of the problem.

Loads from invalid addresses One interesting question is how to handle loads from memory which Memcheck regards as not validly addressable. Our solution is counterintuitive: data loaded from unaddressable memory is marked as defined.

This helps reduce error cascades. A load from an invalid address will in any case cause Memcheck to issue an invalid-address error message. If the loaded data was marked as undefined, Memcheck might, as a result, later issue undefined value error messages. These would confuse users and obscure the true cause of the error—the invalid address. Marking the loaded data as defined avoids that problem.

2.8 Avoiding false positives

Memcheck has a very low false positive rate. However, a few hand-coded assembly sequences, and a few very rare compiler-generated idioms can cause false positives. The few examples we know of are as follows.

- `xor %reg,%reg`: %reg is defined after the instruction, even if it is undefined prior to it. This is solved for Memcheck by Valgrind’s x86-to-UCode translation phase, which translates this idiom as if it had instead seen `mov $0,%reg; xor %reg,%reg`.

- `sbb %reg,%reg`: This copies the carry flag into all bits of `%reg`, and has no real dependence on `%reg`'s original value. The instrumentation described above preserves any undefinedness from `%reg`'s original value, which is inappropriate. Again, the front end solves this by instead translating `mov $0,%reg; sbb %reg,%reg`.
- A more difficult case: GCC occasionally generates code to do a conditional jump based on the highest bit in a register by moving the bit to the sign flag using `test %reg,%reg` and then doing a conditional jump based on the sign flag. Unfortunately, if bits below the highest bit in `%reg` are undefined, Memcheck's instrumentation scheme will conclude that all six condition codes are undefined, and so complain at the jump. The problem arises because only one bit is used to approximate the definedness state of all six condition codes. A possible solution is to model the sign flag separately from the rest, but so far we have resisted this extra complexity and run-time overhead.
- GNU libc contains highly-optimised, hand-written assembly routines for common string functions, particularly `strlen()`. These traverse the string a word at a time, relying on detailed properties of carry-chain propagation for correct behaviour. For such code, Memcheck's use of the `Left` operator to model such propagation is too crude, and leads to false positives.

Memcheck has a two-part work-around. First, it replaces the standard versions of these functions with its own less optimised versions that do not cause problems. But GCC sometimes inlines calls to these functions, and handling them currently involves a nasty hack. Such code requires addition/subtraction of carefully chosen constants, such as `0x80808080`. If Memcheck sees adds/subtracts with such suspect constants as operands, some undefined value checks in the containing basic block are omitted.

A better solution would be to use the presence of such constants as a signal that adds/subtracts in this block should be instrumented using an alternative, more accurate but more expensive formulation which properly tracks carry propagation [6]. We are developing such a scheme.

Finally, Memcheck's underlying assumptions are occasionally invalid. For example, some programs deliberately use undefined values as an additional source of entropy when generating random numbers.

2.9 False negatives

We believe there are very few situations in which Memcheck fails to flag uses of undefined values that could have any observable effect on program behaviour. The exceptions we are aware of are as follows.

- The abovementioned omission of some checks in blocks containing magic constants such as `0x80808080`. This hack could be removed as suggested above, probably with minimal performance loss.
- Caller-saved registers in procedures. Ideally, on entry to a procedure, caller-saved registers should be marked as undefined, since callees assume that caller-saved registers are fresh storage available for use. Memcheck does not currently do so. Doing this correctly is difficult, both because it is calling-convention dependent, and because reliably observing procedure entry/exit on x86/Linux is nearly impossible given the use of tail-call optimisations, leaf-function optimisations, and use of `longjmp()`.⁵ As a result, it is possible that registers which should be regarded as undefined at the start of a callee are marked as defined due to previous activity in the caller, and so some errors might be missed.
- Programs that switch stacks (usually because they implement user-space threading). There is no reliable way to distinguish a large stack allocation or deallocation from a stack-switch. Valgrind uses a heuristic: any change in the stack pointer greater than 2MB is assumed to be a stack-switch. When Valgrind judges that a stack-switch has happened, Memcheck does not take any further actions. So if a stack frame exceeding 2MB is allocated, Valgrind considers this a stack switch, and Memcheck will not mark the newly allocated area as undefined. The program could then use the values in the allocated area unsafely, and Memcheck will not detect the problem.⁶

Finally, Memcheck of course cannot detect errors on code paths that are not executed, nor can it detect errors arising from unseen combinations of inputs. This limitation is inherent from the fact that Memcheck uses dynamic analysis. As a result, Memcheck is best used in conjunction with a thorough test suite. In comparison, static analysis does not suffer these limitations, but the power of the analysis is necessarily much lower [1].

2.10 Setting realistic expectations

A system such as Memcheck cannot simultaneously be free of false negatives and false positives, since that

would be equivalent to solving the Halting Problem. Our design attempts to almost completely avoid false negatives and to minimise false positives. Experience in practice shows this to be mostly successful. Even so, user feedback over the past two years reveals an interesting fact: many users have an (often unstated) expectation that Memcheck should not report any false positives at all, no matter how strange the code being checked is.

We believe this to be unrealistic. A better expectation is to accept that false positives are rare but inevitable. Therefore it will occasionally necessary to add dummy initialisations to code to make Memcheck be quiet. This may lead to code which is slightly more conservative than it strictly needs to be, but at least it gives a stronger assurance that it really doesn't make use of any undefined values.

A worthy aim is to achieve Memcheck-cleaness, so that new errors are immediately apparent. This is no different from fixing source code to remove all compiler warnings, even ones which are obviously harmless.

Many large programs now do run Memcheck-clean, or very nearly so. In the authors' personal experience, recent Mozilla releases come close to that, as do cleaned-up versions of the OpenOffice.org-680 development branch, and much of the KDE desktop environment. So this is an achievable goal.

Finally, we would observe that the most effective use of Memcheck comes not only from ad-hoc debugging, but also when routinely used on applications running their automatic regression test suites. Such suites tend to exercise dark corners of implementations, thereby increasing their Memcheck-tested code coverage.

3 Evaluation

For a tool such as Memcheck to be worth using, a user must first be using a language such as C, C++ or Fortran that is susceptible to the kinds of memory errors Memcheck finds. Then, the benefits of use must outweigh the costs. This section considers first the costs of using Memcheck, in terms of its ease of use and performance. It then considers the benefits, that is, its effectiveness in helping programmers find real bugs.

As part of this, we refer to a survey of Valgrind users that we conducted in November 2003, to which 116 responses were received. The results are available on the Valgrind website [14]. Memcheck's operation has changed very little in the time since, so the responses about it are still relevant.

3.1 Ease of use

Ease of use has a number of facets: how easy Memcheck is to obtain, how easy it is to run, and how easy it is to

act upon its results.

Obtaining Memcheck Memcheck is very easy to obtain, because the Valgrind suite is free software, is available in source form on the Valgrind website [14], and is widely available in binary form on the web, packaged for a number of Linux distributions.

Running Memcheck Memcheck could hardly be easier to run. Using it only requires prefixing a program's command line with `valgrind --tool=memcheck`, plus any other desired Valgrind or Memcheck options.

Typically, the only further effort a user must make is to compile her program with debugging information, to ensure error messages are as informative as possible. Indeed, 40 of the 116 survey responders praised Valgrind/Memcheck's ease of running, and another another 14 commented that "it just works", or similar. Only one responder complained that it could be easier to use.

Custom allocators There are some cases where the user needs to expend a little more effort. If a program uses custom memory management rather than `malloc()`, `new` and `new[]`, Memcheck can miss some errors it would otherwise find. The problem can be avoided by embedding small number of *client requests* into the code. These are special assembly code sequences, encoded as C macros for easy use, that Valgrind recognises, but which perform a cheap no-op if the client is not running on Valgrind. They provide a way for the client to pass information to a Valgrind tool. For example, when using a custom allocator, client requests can be used to inform Memcheck when a heap block has been allocated or deallocated, its size and location, etc.

Self-modifying code Extra effort is also needed to handle self-modifying code. Dynamically generated code is not a problem, but if code that has executed is modified and re-executed, Valgrind will not realise this, and will re-run its out-of-date translations. Auto-detecting this is possible but expensive [7]; the cost is not justified by the small number of programs that use self-modifying code. Our compromise solution is to provide another client request which tells Valgrind to discard any cached translations of code in a specified address range.

Different behaviour under Memcheck Client programs sometimes do not behave exactly the same under Valgrind/Memcheck as they do normally. Programs that run successfully normally may fail under Memcheck. This is usually because of latent bugs that are exposed e.g. by different execution timings, or because Valgrind provides its own implementation of the Pthreads library.

This can be regarded as a good thing. More rarely, programs that fail normally may succeed under Memcheck for the same reasons, which can be frustrating for users who hoped to track down a bug with Memcheck.

Although Valgrind/Memcheck is robust—we have had feedback from users using it on systems with 25 million lines of code—it occasionally fails due to its own bugs. The main source of problems is that Valgrind interacts closely with the kernel and GNU libc. In particular the signal and thread handling is fragile and hard to get right for all Linux distributions, and a maintenance headache. By comparison, the parts dealing with x86 code and instrumentation cause few problems, because instruction sets change very slowly. We hope to decrease our level of interaction with the kernel and GNU libc in the future, and recent development efforts have made major progress in that area.

Acting upon Memcheck’s results In general, Memcheck’s addressability checking, deallocation checking, and overlap checking do quite well here, in that Memcheck’s report of the problem is usually close to the root cause. However, for definedness checking this is often not the case, as Section 2.7 explained.

To help on this front, Memcheck provides another client request that can be inserted into the client’s source code. It instructs Memcheck to check if a particular variable or memory address range is defined, issuing an error message if not. Judicious uses of this client request can make identifying root causes of undefined value errors much easier.

Ideally, error messages would indicate where the undefined value originated from, e.g. from a heap block whose contents have not been initialised. However, this would require augmenting each value with extra information about where it came from, and we cannot see how to do this without incurring prohibitive overheads.

Ease of interpreting error messages is also important. Ten survey responders complained that the messages are confusing, but 7 praised them. One issue in particular is the depth of stack traces: the default is four, but many users immediately adjust that to a much higher number. This gives more information, but also makes the error messages longer. This is a case where a GUI (requested by 8 responders) would be useful, in that large stack traces could be gathered, shown to a small depth by default, and then “unfolded” by the user if necessary. Some users also simply prefer graphical tools over text-based ones. As it happens, there are several GUI front-ends for Valgrind, including Alleyoop [17] and Valgui [2].

Program	t (s)	Mem.	Addr.	Nul.
bzip2	10.8	13.8	10.2	2.5
crafty	3.5	45.3	27.4	7.9
gap	1.0	26.5	19.2	5.6
gcc	1.5	35.5	23.7	9.2
gzip	1.8	22.7	17.7	4.7
mcf	0.4	14.0	7.1	2.6
parser	3.6	18.4	13.5	4.2
twolf	0.2	30.1	20.5	6.1
vortex	6.4	47.9	36.5	8.5
ampp	19.1	24.7	23.3	2.2
art	28.6	13.0	10.9	5.5
quake	2.1	31.1	28.8	5.8
mesa	2.3	43.1	35.9	5.6
median		26.5	20.5	5.6
geo. mean		25.7	19.0	4.9

Table 1: Slow-down factors of Memcheck, Addrcheck and Nulgrind (smaller is better)

3.2 Performance

This section discusses the performance of three Valgrind tools. Besides Memcheck, it considers two other tools: Addrcheck, and Nulgrind. Addrcheck is a cut-down version of Memcheck that does not perform definedness checking. The performance difference between Memcheck and Addrcheck give an indication of the cost of Memcheck’s definedness checking. Nulgrind is the “null” Valgrind tool that adds no instrumentation. It gives an idea of the overhead due to Valgrind’s basic operation.

All measurements were performed using Valgrind 2.1.2 on an 1400 MHz AMD Athlon with 1GB of RAM, running Red Hat Linux 9, kernel version 2.4.20. The test programs are a subset of the SPEC CPU2000 suite [16]. All were tested with the “test” (smallest) inputs. The time measured was the “real” time, as reported by `/usr/bin/time`. Each program was run once normally, and once under each of the Valgrind tools. This is not a very rigorous approach but that does not matter, as the figures here are only intended to give a broad idea of performance.

Table 1 shows the time performance of the three tools. Column 1 gives the benchmark name, column 2 gives its normal running time in seconds, and columns 3–5 give the slow-down factor for each tool relative to column 2 (smaller is better). The first nine programs are integer programs, the remaining four are floating point programs. The bottom two rows give the median and geometric mean for the slow-down factors.

The slow-down figures for Memcheck are quite high. This is partly due to the cost of definedness checking,

partly due to the cost of Memcheck's other kinds of checking, and partly because of Valgrind's inherent overhead.

Memcheck also has a large memory cost. Since each byte is shadowed by a byte holding V bits and also by a single A bit indicating whether that location is addressable, memory use is increased by a factor of approximately $(8 + 1)/8 = 9/8$, that is, slightly more than doubled. Shadow storage is allocated on demand, so programs which do not use much memory when running natively still do not use much when running under Memcheck. Another space cost is that of holding translations: Valgrind can store translations of approximately 200000 basic blocks, occupying about 70MB of storage. This too is allocated incrementally. Finally, Memcheck records on the order of 50 to 100 bytes of administrative information for each block allocated by `malloc/new/new[]`.

As a result of this, programs with large memory footprints (above about 1.5GB) die from lack of address space. Since V-bit storage is usually the largest component of the overhead, we are considering compressed representations for V bits. These rely on the observation that about 99.95% of all bytes are all-defined or all-undefined, and so their definedness state could be summarised using a single bit. This would be merely a representational change and would not affect Memcheck's ability to track definedness with bit-level precision.

Of the various costs to the user of using Memcheck, for many people the slow-down is the greatest cost. Among the survey responders, 9 praised Memcheck's performance, and 32 complained about it. In general, we have found that users who have used other, similar, memory checkers praise Memcheck's performance, and those who have not used other such tools complain about it. However, judging from overall feedback, Memcheck's performance is good enough for the vast majority of users.

3.3 Evidence of usefulness

It is never easy to convincingly demonstrate in a paper that a tool such as Memcheck, which is designed to find bugs in programs, works well. To truly appreciate the usefulness of such a tool, one must really use it "in anger" on a real system. Nonetheless this section provides two pieces of evidence that Memcheck, particularly its definedness checking, is effective. The first is a case study of using it on OpenOffice; the second is general information about Memcheck's popularity.

OpenOffice case study Jens-Heiner Rehtien used Memcheck with OpenOffice⁷, and systematically recorded all the error messages that Memcheck issued [13]. He ran OpenOffice's basic "smoke test" (with

Java disabled), which only exercises a fraction of OpenOffice's code. Memcheck detected 102 problems in 26 source files (plus 6 more in system libraries such as GNU libc). Table 2 (copied from [13]) gives a breakdown: column 1 numbers the problem category, column 2 describes the problem, column 3 gives the number of occurrences, and column 4 gives the number of distinct source files in which the error occurred.

Of the 11 categories, five (1, 2, 3, 7 and 10) involve undefined values. This accounts for 96 of 102, or 94%, of the problems found.

Rehtien estimates that, regarding the consequences of the detected problems, about one third would never show up as a program failure for the user, another third are bugs which have no consequences yet, but might lead to regressions later if code is changed, and the last third are plain bugs which might crash the application or lead to malfunction anytime. The commits made to fix these errors can be seen in the OpenOffice bugs database [11].

As for false positives, Rehtien states that when compiling with optimisation, he saw "a few", but none when compiling without optimisation.

From this example, Memcheck's usefulness is clear, in particular the usefulness of its definedness checking.

Popularity Another indication of the usefulness of Memcheck's definedness checking is its general popularity. We have received feedback from at least 300 users in more than 30 countries, and so can conservatively estimate its user base is in the thousands.

A short list of notable projects using Valgrind includes: OpenOffice, StarOffice, Mozilla, Opera, KDE, GNOME, AbiWord, Evolution, MySQL, PostgreSQL, Perl, PHP, Mono, Samba, Nasa Mars Lander software, SAS, The GIMP, Ogg Vorbis, Unreal Tournament, and Medal of Honour. A longer list is available at the Valgrind website [14]. This includes a huge range of software types, almost anything that runs on x86/Linux.

Our November 2003 survey also found that Memcheck is by far the most commonly used of the Valgrind tools, accounting for approximately 85% of Valgrind use. In comparison, Addrcheck, which is the same as Memcheck but without the definedness checking, only accounts for 6% of Valgrind use.

4 Related work

There are a number of tools that detect various kinds of memory errors in programs, particularly memory leaks and bounds errors. However, only a small fraction of them detect undefined values. This section discusses only work that relates directly to Memcheck's definedness checking.

Ref. no.	Error type	#problems	#files
1	not initialized instance data member	ca. 76	10
2	not initialized local variables	7	5
3	not initialized variable used as in/out parameter in method call	11	3
4	overlapping buffers in strncpy()	1	1
5	off by one error	1	1
6	unchecked return value of system call	1	1
7	partly initialized struct used in inappropriate ways	1	1
8	no check for potentially invalidated index	1	1
9	use of buffer size instead of content size for writing out data	1	1
10	write not initialized buffer into stream	1	1
11	feed unterminated buffer into method expecting a C-style string	1	1

Table 2: Errors found by Memcheck in OpenOffice

Huang [4] described, very early, the possibility of using dynamic analysis to detect variables that are written but never read, and reads of undefined variables. However, the proposed instrumentation was at the source code level rather than the machine code level.

Kempton and Wichmann [5] discussed the detection of undefined value errors in Pascal programs. They suggest using one shadow definedness bit per data bit in memory, just as Memcheck does. They did not discuss how the shadow bits would be propagated.

Purify [3] is a widely used commercial memory checking tool that can detect several kinds of memory errors including undefined value errors. It adds instrumentation to object code at link-time, in order to do checking at run-time. It shadows every byte of memory with a two-bit value that encodes one of three states: unaddressable, writable, and readable. Reads of writable (i.e. allocated but undefined) memory bytes may get flagged immediately as errors. This is a form of eager checking, as mentioned in Section 2.6. Because the shadowing is at the byte-level, it does not feature bit precision.

Third Degree [18] is a memory checking tool built using the ATOM object code instrumentation framework [15]. As well as detecting some accesses to unaddressable memory, it uses a crude form of definedness checking: when a stack or heap location first becomes addressable, it is written with a special “canary” value. An error is issued if any loads read this canary value. This could lead to false positives when undefined values are copied around, and there is a small chance that the canary value might occur legitimately, although the value is chosen carefully to minimise this likelihood. As Third Degree only ran on Alphas, it is unfortunately now defunct.

Insure++ [12] is a commercial memory checking tool. It detects various kinds of errors, including undefined value errors. It adds instrumentation to the source code (C or C++ only) at compile-time, in order to do checking at run-time. It has two modes of undefined-

ness checking: the first is eager, immediately reporting any reads of undefined variables; the second is less eager, allowing copies of undefined variables to occur without warning (but not other operations). Insure++ also comes with Chaperon, a tool for x86/Linux that works much like Memcheck—it also uses dynamic binary instrumentation—which can detect reads of uninitialised memory.

The most notable feature of Memcheck compared to previous tools is that its definedness checking is much more sophisticated, featuring bit-precision, which means it can be used reliably with bit-field operations.

5 Conclusion

We have described a tool named Memcheck, built with the dynamic binary instrumentation framework Valgrind, which can detect several kinds of memory errors that are common in imperative programs. In particular, we described the novel definedness checking that Memcheck performs to detect undefined value errors with bit-precision. We also considered the costs and benefits of using Memcheck, and concluded that it definitely is worth using, as shown by the 102 bugs found in OpenOffice by one user, and more generally by the thousands of programmers who use it on a wide range of software.

Memcheck works well, and is widely used. Our main challenge is ensuring that this remains true. Work is in progress to develop a new intermediate representation for Valgrind, which will provide sufficient description of FP and SIMD operations to keep up with the increasing capabilities of vectorising compilers. The new intermediate representation will also be architecture-neutral, in order to provide a solid foundation for ports to architectures other than x86, which will help keep Valgrind viable over the long-term. A key challenge here is that instrumentation should also be architecture-neutral, so that tools do not have to be partially or wholly re-written for each new

architecture supported. We also are working on ports to other operating systems, in order to remove Valgrind's dependence on Linux. This is a significant task, but one that is orthogonal to Memcheck's workings.

If you are developing programs in C, C++, Fortran or Pascal on the x86/Linux platform, you almost certainly should be using Memcheck. If you are not, all the bugs Memcheck would find will have to be found manually, wasting your development time, or not be found at all, compromising your code quality.

6 Acknowledgments

Thanks to Donna Robinson for encouragement, and Jens-Heiner Rehtien for his meticulous record-keeping. The second author gratefully acknowledges the financial support of Trinity College, University of Cambridge, UK.

Notes

¹However, it is useful mostly for languages like C, C++ and Fortran, which have scant protection against memory errors.

²Purify 2003a.06.13 Solaris 2 (32-bit, sparc) running on Solaris 2.9. Compiler was Sun C 5.5. Optimisation level makes no difference: we tried -xO3, -xO5 and no optimisation, with the same result.

³Use of an undefined address is different from use of a defined but invalid address. Memcheck detects both kinds of errors, but only the former is the subject of this discussion.

⁴Or shadow memory byte, for memory arguments to system calls.

⁵A common question from aspiring Valgrind hackers is how to write a simple tool which observes procedure entries/exits. From experience we know that doing so reliably is extraordinarily difficult. Josef Weidendorfer's "Callgrind" tool [19] is the best attempt so far, and it is surprisingly complex.

⁶For similar reasons, the stack-switch heuristic can also confuse Memcheck's addressability checking.

⁷On the 680 branch, which will become OpenOffice 2.0.

References

- [1] Michael D. Ernst. Static and dynamic analysis: synergy and duality. In *Proceedings of WODA 2003*, pages 6–9, Portland, Oregon, May 2003.
- [2] Eric Estievenart. Valgui, a GPL front-end for Valgrind. <http://valgui.sf.net/>.
- [3] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, San Francisco, California, USA, January 1992.
- [4] J. C. Huang. Detection of data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering*, 5(3):226–236, May 1979.
- [5] Willett Kempton and Brian A. Wichmann. Runtime detection of undefined variables considered essential. *Software—Practice and Experience*, 20(4):391–402, April 1990.
- [6] Paul Mackerras. Re: Valgrind for PowerPC. Message to the valgrind-developers mailing list, March 2004.
- [7] Jonas Maebe and Koen De Bosschere. Instrumenting self-modifying code. In *Proceedings of AADE-BUG2003*, Ghent, Belgium, September 2003.
- [8] Dorit Naishlos. Autovectorisation in GCC. In *Proceedings of the 2004 GCC Developers' Summit*, Ottawa, Canada, June 2004.
- [9] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, November 2004.
- [10] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of RV'03*, Boulder, Colorado, USA, July 2003.
- [11] Openoffice.org issue 20184, 2003. http://www.openoffice.org/issues/show_bug.cgi?id=20184.
- [12] Parasoft. Automatic C/C++ application testing with Parasoft Insure++. White paper.
- [13] Jens-Heiner Rehtien. Validating and debugging openoffice.org with valgrind, 2003. <http://tools.openoffice.org/debugging/usingvalgrind.sxw>.
- [14] Julian Seward, Nicholas Nethercote, Jeremy Fitzhardinge, et al. Valgrind. <http://www.valgrind.org/>.
- [15] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of PLDI '94*, pages 196–205, Orlando, Florida, USA, June 1994.
- [16] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmarks. <http://www.spec.org/>.
- [17] Jeffrey Stedfast. Alleyoop. <http://alleyoop.sf.net/>.
- [18] *Third Degree User Manual*, May 1994.
- [19] Josef Weidendorfer. KCachegrind. <http://kcachegrind.sf.net/>.

Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution

Tong Li¹, Carla S. Ellis¹, Alvin R. Lebeck¹, and Daniel J. Sorin²

¹*Department of Computer Science*

²*Department of Electrical and Computer Engineering
Duke University*

{tongli, carla, alvy}@cs.duke.edu, sorin@ee.duke.edu

Abstract

Deadlock can occur wherever multiple processes interact. Most existing static and dynamic deadlock detection tools focus on simple types of deadlock, such as those caused by incorrect ordering of lock acquisitions. In this paper, we propose *Pulse*, a novel operating system mechanism that dynamically detects various types of deadlock in application programs. Pulse runs as a system daemon. Periodically, it scans the system for processes that have been blocked for a long time (e.g., waiting on I/O events). To determine if these processes are deadlocked, Pulse speculatively executes them ahead to discover their dependences. Based on this information, it constructs a general resource graph and detects deadlock by checking if the graph contains cycles. The ability to look into the future allows Pulse to detect deadlocks involving consumable resources, such as synchronization semaphores and pipes, which no existing tools can detect. We evaluate Pulse by showing that it can detect deadlocks in the classical dining-philosophers and smokers problems. Furthermore, we show that Pulse can detect a well-known deadlock scenario, which is widely referred to as a denial-of-service vulnerability, in the Apache web server. Our results show that Pulse can detect all these deadlocks within three seconds, and it introduces little performance overhead to normal applications that do not deadlock.

1 Introduction

Concurrent programs are difficult to write and debug. One common problem is deadlock. Deadlock can occur wherever multiple processes (or threads) interact. A set of processes is deadlocked if each process is waiting for an event that only another process in the set can cause. Deadlock is a potential problem in all multithreaded programs. Timely detection of deadlock and its cause is essential for resolving the error and maintaining forward progress.

To address this problem, researchers have developed deadlock detection mechanisms. In practice, however, deadlock detection is often not performed in an effective manner. The drawbacks of the various existing approaches include restrictions on the deadlock-prone access patterns that can be handled and lack of information provided on the causes of deadlocks that arise. We classify existing approaches based on whether they are dynamic or static.

One common dynamic deadlock detection approach is to use timeouts. With timeouts, a process is assumed deadlocked after waiting for a shared resource longer than a certain amount of time. This approach is simple, but inaccurate because it cannot differentiate between processes that are deadlocked and processes that simply need a long time to acquire a resource. Even when they detect deadlock correctly, timeouts provide no information to the developer about why the deadlock occurred.

An alternative dynamic approach is based on graph modelling of process interactions. The “textbook” deadlock detection uses the general resource graph model [9], which models the state of a system as a directed graph. The nodes in the graph represent processes and resources, and the edges represent dependences among them. Given a general resource graph, deadlock can be detected by checking if the graph possesses certain properties (e.g., a cycle or a knot). The general resource graph model classifies resources into reusable and consumable. A reusable resource has a fixed number of units; one unit can be assigned to at most one process at

a time. A consumable resource has no fixed total number of units; when a unit is assigned to a process, it ceases to exist. Only a process that is designated as a producer of a consumable resource can produce units of the resource.

In practice, deadlock detection often assumes a simplified resource model: the system contains only reusable resources and there is only a single unit of every resource. This model makes deadlock detection simple to implement, but at the cost of detecting fewer types of deadlock. Under this model, a general resource graph takes a much simpler form as a wait-for-graph (WFG). The nodes in a WFG represent processes and the edges represent dependences between processes—there is an edge from node A to node B if process A is waiting for process B to release a resource. A cycle in a WFG indicates a deadlock. Constructing a WFG requires dynamically tracking the status of the resources. This includes tracking the owner of each resource and the processes that are waiting for the resource at any time.

A common disadvantage of all dynamic techniques is that their analysis only considers control flow paths actually taken. Static deadlock detection (e.g., RacerX [5]) does not have this problem because it performs analysis on all possible control flow paths. However, these methods depend upon programmer specification of lock semantics and availability of the entire code base. Considering all possible paths also forces static tools to face the issue of filtering out potentially large amounts of false positives.

In this paper, we propose *Pulse*, an operating system mechanism that dynamically detects deadlock. Pulse is based on the general resource graph model. It uses high-level speculative execution [3, 6] to construct dependency information about processes that are blocked in the OS kernel. Throughout this paper, we use the terms *processes* and *threads* interchangeably to refer to the basic units of scheduling. Our goal is to detect a wide variety of deadlock situations, including those that can and cannot be detected by existing techniques. However, our intent is not to replace existing techniques, but to increase the types of deadlock that can be detected by developers. Pulse is complementary to existing techniques; when Pulse and the other tools are used together, they can provide the best coverage of deadlocks.

Our implementation of Pulse focuses on detecting deadlocks caused by bugs in application programs as opposed to bugs in kernel code. We have implemented Pulse in Linux kernel version 2.6.8.1. Our results show that Pulse can detect deadlock situations in incorrect solutions to the classical dining-philosophers and smok-

ers problems, and a deadlock scenario in the Apache web server version 2.0.49.

Pulse runs as a daemon process and performs deadlock detection within the OS kernel when necessary. Pulse can be in one of three modes: *nap*, *monitor*, and *detection*. Initially, it is in the nap mode (i.e., the Pulse process sleeps in the kernel). Periodically, it awakens and enters the monitor mode. In this mode, Pulse checks if any process in the system has been asleep for a long time (how long is a tunable parameter). If none is found, Pulse returns to the nap mode. Otherwise, the sleeping processes might be deadlocked, and thus Pulse enters the detection mode. In this mode, Pulse identifies the events on which these long sleeping processes are waiting (e.g., a lock being free or a pipe being non-empty). To discover how these long sleeping processes depend on each other, Pulse forks each of them to create a *speculative process*. A speculative process first modifies its state such that it will not block again (e.g., by setting the status of the lock on which its parent is blocked to free in its own address space), and then executes ahead in its parent's program. To prevent a speculative process from changing the state of normal processes, Pulse leverages the copy-on-write mechanism in Unix fork and disallows a speculative process to perform I/O writes [6].

During the execution of a speculative process, Pulse records all the events it creates (e.g., releasing a lock or writing to a pipe). These events are then used to match against the events awaited by the sleeping processes. A match between processes *A* and *B* indicates that if process *A* were unblocked, it would produce an event that unblocks process *B*. Based on this information, Pulse constructs a general resource graph in which the nodes denote the sleeping processes and the events on which they are waiting, and the edges denote the dependences Pulse discovers. Pulse detects deadlock by checking if this graph contains cycles. If it detects a cycle, it also prints out the entire graph to help programmers identify the causes of the deadlock.

Pulse differs from existing dynamic techniques in that it discovers dependency information by looking into the future, while existing techniques rely on past information (e.g., who owns a lock and who is waiting for it) to derive the dependences. Most existing techniques are WFG-based and they commonly restrict themselves to only detect deadlocks involving single-unit reusable resources such as locks. Fundamental in this resource model is the assumption that a busy resource can only be freed by the owner that currently holds the resource. This assumption makes it possible to discover process dependences based on information collected from the past, which includes the identities of the owners and

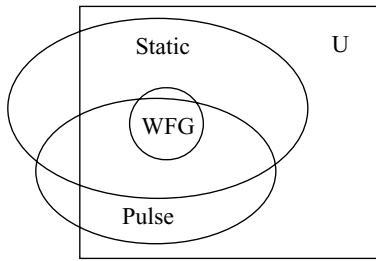


Figure 1: Venn diagram for deadlocks detectable by WFG-based techniques, static schemes, and Pulse. U is the universal set of all deadlocks. The regions outside the rectangle represent false positives.

waiters of each resource. However, consumable resources cannot be viewed as held by a process. For example, a semaphore used for synchronization (instead of mutual exclusion as a lock) may not be viewed as being held by a single process—any process can potentially perform an up operation on the semaphore and unblock another waiting process. In contrast, Pulse makes no assumption about the resource model. The ability to look into the future allows Pulse to directly discover what events a process can produce, as opposed to reasoning about it based on ownership information.

The ability to detect deadlocks that involve consumable resources also distinguishes Pulse from existing static approaches, such as RacerX [5]. In Figure 1, we use a Venn diagram to illustrate the types of deadlock that WFG-based techniques, static schemes, and Pulse can detect. We draw the set of deadlocks detectable by the static schemes larger than the other sets, because static schemes detect deadlocks along all possible control flow paths, while WFG-based techniques and Pulse only detect deadlocks that occur during execution. There is one particular type of deadlock that Pulse cannot detect while WFG-based techniques and static schemes can. This happens when Pulse cannot discover all the dependences from running ahead in the program. For example, if a sloppy programmer forgets an unlock operation that can unblock a waiting process, then Pulse will not see this future event and thus will not be able to identify a cycle of dependences.

There are also types of deadlock that Pulse can detect but the other approaches cannot. These include deadlocks involving consumable resources (e.g., RacerX ignores deadlocks with synchronization semaphores) and variable aliasing (e.g., different variables may point to the same lock, which may not be detectable with static analysis). On the other hand, both static detection and Pulse can generate false positives, i.e., detect deadlocks that do not really exist. Static detection can generate more false positives because it cannot completely

filter out control flow paths that are never taken in real execution. However, Pulse does create a unique set of false positives that other techniques do not encounter, which we discuss in Section 3.5.

Pulse can be viewed as complementary to the existing deadlock detection techniques. If Pulse and the other techniques are used together, they can provide the best coverage of deadlocks.

The remainder of this paper is organized as follows. In Section 2, we discuss related work. Section 3 describes how Pulse works and its limitations. We discuss how we implement Pulse in Section 4. In Section 5, we demonstrate that Pulse can detect deadlock situations in the classical dining-philosophers and smokers problems, as well as in the Apache web server. Existing techniques, including WFG-based schemes and RacerX, can only detect deadlock in the dining-philosophers problem, but cannot detect the other two deadlock situations. Finally, we conclude in Section 6.

2 Related Work

In this section, we provide more details about related work on dynamic and static deadlock detection and OS-level speculative execution.

Dynamic deadlock detection. Most dynamic schemes detect simple deadlocks that involve only lock-like resources. By tracking every acquire and release of the shared resources, these schemes can discover cyclic dependences among the processes. There is a large body of research on deadlock detection in distributed systems (see a survey by Singhal [11]). Some software systems also provide dynamic deadlock detection functionalities. For example, the Windows Driver Verifier and Java HotSpot VM both can track the use of locks and detect cyclic lock dependences. Database systems, such as Berkeley DB, MySQL, Oracle, and PostgreSQL, use timeouts and WFGs to detect deadlock. The Linux kernel can perform simple deadlock detection whenever the `fcntl` system call is invoked. Havelund [8] describes a dynamic deadlock detection mechanism as an extension to NASA’s Java Pathfinder 2 model checking system [13]. This mechanism records lock operations executed by each Java thread and performs post-mortem analysis to detect potential deadlocks. Different from all these mechanisms, Pulse does not assume only lock-like resources and can detect more general forms of deadlock.

Static deadlock detection. Model checking is a formal verification technique that searches in a program’s state space for possible errors, including deadlock. Example model checking systems include Bandera [4], VeriSoft

[7], SPIN [10], and Java Pathfinder [13]. However, model checking large, complex software systems is still impractical due to the state explosion problem.

Microsoft suggests modelling multithreaded Win32 applications as Petri Nets and using their DLDETECT tool to statically analyze programs for potential deadlock [1]. Sun Solaris provides the Crash Analysis Tool (CAT) that helps users statically analyze system crash dumps to identify simple lock-induced deadlocks. Similarly, Linux supports the Non-Maskable Interrupt (NMI) watchdog, which periodically prints out system information that can help statically identify deadlock.

Recently, Engler et al. [5] proposed RacerX, a static tool for checking data races and deadlocks in large software systems. RacerX annotates source code of the system being checked, constructs the whole-system control flow graph, and searches within the graph for possible deadlocks. An important part of RacerX is to rank the detected deadlocks in terms of how likely they are to occur and filter out inaccurate deadlock warnings. RacerX can only detect deadlocks involving lock-like resources. In contrast, Pulse can detect more complex deadlocks involving consumable resources.

Speculative execution. Chang and Gibson [3] proposed a design for automatically transforming applications to perform speculative execution and issue hints for their future I/O read accesses. Fraser and Chang [6] improved this design by leveraging existing OS features to perform speculative execution. To ensure safety, they sever the ordinary relationships between speculative processes and their parents, and disallow speculative processes to execute potentially unsafe system calls. Pulse employs similar techniques to ensure safety.

3 Deadlock Detection with Pulse

In this section, we present an overview of Pulse (Section 3.1) and then describe its design in detail (Section 3.2–Section 3.4). Finally, we discuss how Pulse can be extended and its current limitations (Section 3.5). Section 4 presents our Linux implementation of Pulse.

3.1 Overview

Pulse exploits the observation that if a set of processes is deadlocked, the processes often sleep within the OS kernel, each waiting for events that can only be produced by another process in the set. Thus, when Pulse sees a set of processes blocked for a long time, it considers that a deadlock might have occurred.

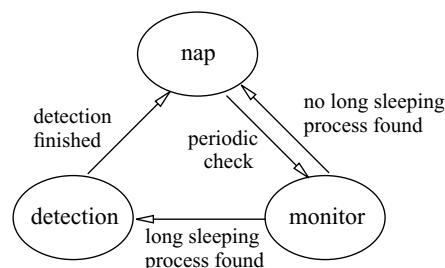


Figure 2: Pulse state transition diagram.

Process P_1	Process P_2
lock(A)	lock(B)
lock(B)	lock(A)
unlock(A)	unlock(B)

Figure 3: A circular lock example.

Pulse runs as a daemon process that can be in one of three modes: nap, monitor, and detection. Figure 2 is a state diagram that shows how Pulse transitions between these modes. For most of the time, Pulse is in the nap mode in which it sleeps in the OS kernel. Periodically, it awakens and enters the monitor mode to check if any process in the system has been asleep for a threshold amount of time, where the threshold value is a tunable parameter (e.g., five minutes). If no such processes are found, Pulse returns to the nap mode, thus incurring low overhead. Otherwise, Pulse enters the detection mode and performs deadlock detection for these sleeping processes. We show in Section 5.4 that a threshold as low as one minute introduces negligible performance overhead. However, if the threshold is too large, Pulse may miss certain deadlocks that can be broken by mechanisms such as timeouts, as we explain in Section 3.5.

Pulse uses the general resource graph model [9] to detect deadlock. To illustrate the idea, we use the code in Figure 3 as a running example. Suppose that the two processes execute their second `lock` statements simultaneously. Thus process P_1 blocks on lock B and process P_2 blocks on lock A , and they deadlock. When Pulse detects that processes P_1 and P_2 have been asleep longer than the threshold, it enters the detection mode.

In the detection mode, Pulse identifies the events on which the sleeping processes are waiting (Section 3.2). In our example, process P_1 is waiting for lock B to be free and process P_2 is waiting for lock A to be free. For each sleeping process, Pulse constructs a *process node* in a general resource graph. For each event it identifies, Pulse constructs an *event node*. We use the term event node instead of resource node [9] to emphasize that a

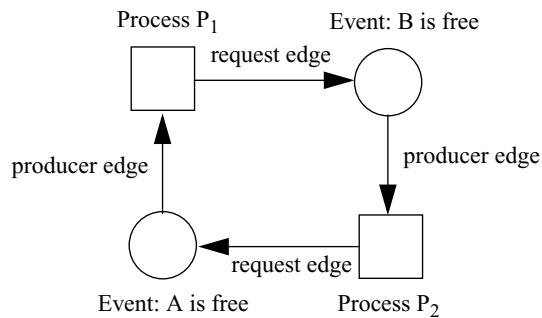


Figure 4: Resource graph for the circular lock example.

process often waits for an event involving a resource (e.g., a lock being free), as opposed to the resource itself (e.g., the lock). Pulse also constructs a *request edge*, directed from a process node to an event node, if the process is waiting for that event.

To discover dependences between the sleeping processes, Pulse uses speculative execution. For each sleeping process, Pulse forks a speculative process that executes ahead in its parent's program. Speculative execution allows Pulse to discover the events that a blocked process would produce if it were not blocked. In our example, Pulse discovers that process P_1 would unlock A and process P_2 would unlock B , if they were not blocked. Thus Pulse constructs a *producer edge*, directed from the event that process P_1 creates, i.e., lock A becoming free, to process P_1 . Similarly, Pulse constructs a producer edge from the event node that P_2 creates, i.e., lock B being free, to process P_2 .

Figure 4 shows the graph that Pulse finally obtains. We use squares to denote process nodes and circles to denote event nodes. Since this graph contains a cycle, Pulse outputs that deadlock exists. It also outputs the entire graph to help developers debug the deadlock.

The general resource graph model allows us to detect whether a deadlock exists, and if so, which processes are involved in the deadlock and why they are deadlocked. To construct a general resource graph, our design needs to address the following questions:

1. How do we construct nodes in the graph? In other words, how do we identify the processes and events involved in a potential deadlock? (Section 3.2)
2. How do we construct edges in the graph? That is, how do we identify the dependences among the processes and events? (Section 3.3)

In the rest of this section, we describe how our design addresses these questions in detail.

3.2 Constructing nodes

When Pulse enters the detection mode, it has already identified a set of processes that have been asleep for a long time. These processes are potentially deadlocked; thus, they constitute the process nodes in the general resource graph.

The events on which these processes are blocked constitute the event nodes in the graph. To identify the events for which a sleeping process is waiting, Pulse requires all blocking system calls to be modified. Within these calls, we add new code to record the events for which the caller process is waiting. Pulse characterizes an event by a *(resource, condition)* pair. The *resource* field identifies the resource on which the process is blocked (e.g., a lock or a pipe). The *condition* field describes the condition about the resource for which the process is waiting (e.g., the lock being free, or the pipe being non-empty).

3.3 Constructing edges

Pulse models all resources as consumable resources. This resource model enables Pulse to detect deadlocks involving more than just single-unit reusable resources. However, it also causes Pulse to generate certain false positives, as we will see in Section 3.5.

There are two sets of edges that Pulse needs to construct in a graph: request edges and producer edges. The request edges can be constructed at the same time when Pulse constructs the event nodes. As discussed in Section 3.2, when Pulse identifies a sleeping process and its awaited events, it can construct a request edge from the process to each event for which it is waiting.

To construct the producer edges, Pulse uses speculative execution. For each sleeping process, Pulse forks a copy of the process, called a *speculative process*. The speculative process first creates the events awaited by its parent process, if this does not affect any other process. For example, if the parent is blocked on a busy lock, then the speculative process can set the lock variable (often a user-space memory location) to free within its own address space, not affecting any other process. On the other hand, if the parent has been waiting for an I/O, the speculative process is not allowed to create the awaited event because it could affect the state of other processes. Regardless of whether the events are created or not, the goal of Pulse here is to enable the speculative process to unblock and thus run ahead in its program. The next step for the speculative process is to return from the system call that caused its parent to sleep, pretending the call was successful, and then to resume its

parent's user-level code after the blocking system call, all within the speculative process's own context (i.e., the parent is unchanged and it continues to sleep).

The execution of a speculative process should be safe, i.e., it should not modify the state of any other process. The Unix fork mechanism implements copy-on-write, which naturally protects a speculative process from modifying the user-space memory state of its parent (and other processes). Some systems support forking a process (or thread) that shares the same address space with the parent. Pulse should avoid using fork in such a way and it should always invoke fork with copy-on-write enabled. Similar to Fraser and Chang [6], we do not allow a speculative process to write to the file system (thus no I/O writes) or deliver a signal to any other process. In Section 4, we discuss in detail the extra measures that our implementation takes to ensure the safety of the kernel state. These safety measures, however, may cause Pulse to produce false positives and/or false negatives of deadlock, as we explain next.

During the execution of a speculative process, Pulse records all the events produced by the process that could potentially unblock other processes. This requires modifying the system calls that counterpart the blocking system calls. For example, a write system call can block when trying to write to a full pipe (i.e., the pipe buffer is full), and can be unblocked only if a process reads data from the pipe. Thus, a pipe read system call counterparts a blocking pipe write system call. We modify all system calls that can unblock a process such that, if called by a speculative process, they record the resources being manipulated and the conditions being produced by the speculative process. As for the pipe example, the modified read system call would record a unique resource identifier for the pipe and some indicator representing that the pipe is being read. For each speculative process, Pulse maintains an *event buffer* that records the events the process produces during its execution. A speculative process adds an event to its event buffer only if the event is not already in the buffer. If the buffer is full, the process ignores the event (i.e., does not add it to the buffer). As we see in Section 5, a buffer of ten events is sufficient for all of our experiments.

A speculative process terminates if one of the following conditions is true:

1. It exits normally.
2. Its event buffer is full.
3. T seconds have passed since the creation of the speculative process, where T is adjustable by the user.

After all the speculative processes terminate, Pulse matches their produced events against the events awaited by the sleeping (parent) processes. If the speculative version of process A produces an event that process B is waiting for, then Pulse constructs a producer edge from this event to process A . A speculative process could produce events that are not awaited by any sleeping process. We do not include these events in the graph, because this reduces the graph size and does not affect the correctness of deadlock detection.

3.4 Putting it all together

The nodes and edges that Pulse constructs collectively form a general resource graph. If the graph contains cycles, Pulse outputs that a deadlock exists. To facilitate debugging, Pulse also prints out the entire resource graph. It is also possible to perform symbol table look-ups such that the application programmer can map the graph to specific points in the code. Based on all this information and knowledge of the applications, the programmer could easily verify whether a deadlock indeed exists, and, if so, identify its cause.

Pulse requires OS kernel support (e.g., for speculative execution). Compared to user-space solutions, Pulse provides a general solution for all applications, thus freeing programmers from the burden of designing deadlock detection for each individual application. On the other hand, for applications that already have some deadlock detection built-in, they can use Pulse together with their own mechanisms to obtain the best coverage of deadlock.

3.5 Discussion

In this section, we first briefly describe our design plans for handling deadlocks involving spinning processes and deadlocks due to kernel bugs, but we leave a full evaluation of these designs as future work. We then discuss in detail the limitations of Pulse.

Spin deadlocks. A process can wait for synchronization events via spinning (a.k.a., busy-waiting). Although most commercial software puts a waiting process to sleep (possibly after spinning for a short period), spinning can be prevalent in scientific applications. To detect deadlocks involving spinning processes, we need to dynamically identify the spinning processes and the events for which they are waiting. We can achieve this by instrumenting synchronization libraries. After this information is obtained, Pulse can detect deadlock in the same way as it does for sleeping processes.

Kernel deadlocks. Applying Pulse to detect deadlocks due to kernel bugs is difficult, because allowing processes to speculatively execute within the kernel can cause unwanted changes to kernel structures and even crash the system. However, with the help of virtual machine technologies, such as VMware [2] or Xen [14], we could perform speculative execution on different virtual machines, making Pulse possibly applicable to detecting kernel deadlocks.

Limitations of Pulse. Pulse can output false positives, i.e., deadlocks that do not actually exist. There are three reasons why false positives can occur. First, because the execution of a speculative process must be safe, it cannot perform potentially unsafe operations, such as writing to a file. This could cause a speculative process to execute unrealistic program paths, making Pulse obtain incorrect dependences. Such false positives also exist in static detection tools because they often cannot identify unrealistic program paths statically.

Second, as we discussed in Section 3.3, Pulse models all resources as consumable resources. Thus it could construct more than one producer edge for resources (e.g., locks) that can be held and freed by only a single owner. The extra edges could cause Pulse to detect cycles that do not actually exist. Such false positives cannot occur in existing static and dynamic detection tools because they model only reusable resources.

Third, for systems with resources more complex than single-unit reusable resources, a cycle in a general resource graph is only a necessary, but not sufficient condition for deadlock [9]. For example, Pulse may detect a cycle when multiple processes block on a set of synchronization semaphores. However, a new process may later enter the system and perform an up operation on one of the semaphores, thus breaking the “deadlock”. Such false positives are unique for Pulse; all the existing approaches do not consider resources more complex than single-unit reusable resources and thus do not have such false positives.

There are two types of deadlock that Pulse cannot detect (i.e., false negatives). First, some applications employ a timeout mechanism on operations that take too much time. For example, as we will see in Section 5, Pulse can detect a deadlock scenario due to cyclic pipe access dependences in the Apache web server. However, Apache can abort the pipe operations after they take longer than five minutes. The timeout effectively breaks the deadlock, although it also silently fails the client’s request without providing any information about what has happened. Pulse could miss such deadlocks if its threshold value for entering the monitor mode is too large. However, such false negatives can be avoided by

adjusting the threshold, possibly at the cost of impacting application performance.

The second type of false negative is due to Pulse’s reliance on the future events it discovers. Pulse is able to detect deadlock because it can discover the future events that the sleeping processes could produce if awakened. However, if such events are unavailable, Pulse will not detect the deadlock. There are four scenarios in which the future events can be unavailable.

1. Such events do not exist in the application program. For example, the programmer forgets the unlock statements in the code in Figure 3.
2. Speculative processes do not run long enough to discover the events.
3. The event buffers fill up before speculative processes see the relevant events.
4. Speculative processes may execute unrealistic program paths that do not manifest the relevant events.

The first scenario is a fundamental limitation of Pulse. However, the second and third scenarios are not fundamental limitations; they can be avoided by increasing the run time and event buffer sizes of speculative processes. The reason for the fourth scenario is that Pulse does not allow speculative processes to execute potentially unsafe system calls. This scenario may be considered as a fundamental limitation of Pulse, if our implementation chooses to run speculative processes on the same operating system on which the normal processes run (which is what we do in this paper). However, we may avoid some cases of this scenario if each speculative process can run on a different operating system, e.g., using VMware [2] or Xen [14].

4 Implementation

In this section, we discuss how we implement the design described in the previous section. We implement Pulse by modifying Linux kernel version 2.6.8.1. We add a new system call that allows Pulse to be invoked from the user level.

4.1 Constructing process nodes

To construct the process nodes, Pulse needs to identify long sleeping processes. We add a flag, `was_asleep`, to each process’s `task_struct`, which is set to false when the process is created. When Pulse enters the monitor mode, it scans the system for processes that satisfy the following three conditions:

- The process is asleep.

- The process was put to sleep by one of our modified system calls (see Section 4.2).
- The process's `was_asleep` flag is true, which indicates that the process was asleep when Pulse checked it last time.

If a process satisfies the first two conditions, but not the last one, Pulse sets the process's `was_asleep` flag to true. This flag is reset to false when the Linux scheduler switches this process to run, i.e., when it is awakened. For each process that satisfies all the three conditions, Pulse constructs a process node for it.

Some system daemon processes (e.g., `automount`) sleep in the kernel for a long time. If they satisfy the above conditions, Pulse will construct process nodes for them. Alternatively, we could implement a mechanism to disable the construction of nodes for these processes, if the user of Pulse (e.g., a system administrator) believes that these processes do not deadlock.

4.2 Constructing event nodes

To identify the events for which a sleeping process is waiting, we need to modify all system calls that can block. For the purposes of this paper, we have modified only three blocking system calls: `futex`, `write`, and `poll`. In each call, before putting the caller to sleep, we add code to construct the following two lists and store them in a structure pointed to by the caller process.

- A resource list, (`resource1`, `resource2`, ...), where `resourcei` is an integer that uniquely identifies a resource being manipulated by the system call.
- A condition list, (`<op1, val1>`, `<op2, val2>`, ...), where the pair `<opi, vali>` encodes the condition about `resourcei` that can unblock the caller process.

The `futex` and `write` system calls involve only one resource and thus their resource and condition lists consist of only one element. The `poll` system call, however, can operate on multiple file descriptors. Thus it may record more than one resource and condition. We now describe how we modify these three system calls.

Futex. `Futex` stands for fast user-space mutex. The `futex` system call is the basis of various synchronization primitives in the Native POSIX Thread Library (NPTL), which has been integrated in the recent versions of `glibc`. For the purposes of demonstrating Pulse, we consider NPTL's mutex and semaphore primitives in `glibc` 2.3.2. A thread acquiring a busy mutex or semaphore blocks via the `futex` system call. Each NPTL mutex or semaphore corresponds to a user-space memory address, which is passed to `futex` as an argument. Thus, in `futex` (before the caller is about to sleep), we add

code to record this memory address, which uniquely identifies the resource on which the caller thread is blocked. The condition to unblock the caller, however, depends on the context in which the `futex` system call is invoked, and thus requires modifications to the NPTL library. Note that the applications using the library do not need to be modified.

For the NPTL function that acquires a mutex lock, (`pthread_mutex_lock`), we pass `<equal-to, 0>` as two extra arguments to the `futex` system call. They signify that the condition to unblock the caller thread is when the mutex value is zero (i.e., the mutex is free).

For the NPTL function that does a semaphore down operation (`sem_wait`), we modify it to pass `<greater-than, 0>` as two extra arguments to the `futex` system call. They signify that the condition to unblock the caller is when the semaphore value is greater than zero.

Write. Linux's `write` system call is a generic interface to a wide range of file systems. Our implementation currently considers only writes to pipes, which is implemented in the `pipe_writew` function. For a blocking pipe write, the caller process blocks if the pipe buffer is full. We add code in function `pipe_writew` (before the caller is about to sleep) to record the address of the pipe's inode structure, which uniquely identifies the pipe resource. The condition under which the caller unblocks is when another process reads data from the pipe. We use the pair `<and, POLLOUT | POLLWRNORM>` to encode this condition. The fields `POLLOUT` and `POLLWRNORM` are kernel-defined bit masks, denoting that writing now becomes unblocked. As we will see in Section 4.3, we also record similar bit masks in system calls that can unblock the write to denote the events produced by those calls. The `and` operator here helps match the blocked process with the process that can potentially unblock it: if the logical `and` of `POLLOUT | POLLWRNORM` and the bit mask produced by another process is true, then that process can unblock this process.

Poll. The `poll` system call takes a list of file descriptors and events as arguments. If none of the events has occurred for any of the file descriptors, the caller blocks, waiting for one of these events to occur. The events are represented by bit masks, similar to the ones we discussed above. We add code in the `poll` system call (before the caller is about to sleep) to construct a resource list. Each resource is a file descriptor, denoted by the address of its inode structure. We also record a condition list. Each condition is denoted by a `<op, val>` pair, where `op` equals `and`, similar to what we do for pipe writes. The `val` field is simply the event bit mask that the caller passed to the `poll` system call.

Based on the information recorded by the modified system calls, Pulse can identify what events a sleeping process is waiting for and create the event nodes for them. The three modified system calls allow us to demonstrate the ability of Pulse in this paper. In general, the same approach can be applied to modify other blocking system calls.

4.3 Constructing edges

As we discussed in Section 3.3, Pulse can construct request edges at the same time when it constructs the process and event nodes. Thus, in this section, we focus on constructing producer edges via speculative execution. We explain our implementation by following the flow of speculative execution in time.

Creating speculative processes. After Pulse identifies the long sleeping processes, it forks a speculative process for each of them. The Unix fork mechanism allows a speculative process to run in its own address space, not affecting the state of its parent. The difficulty, however, is that the existing fork implementation assumes its caller process to be the same as the one to be forked. What we need, instead, is a function that can fork an arbitrary given process. Our first attempt was to write such a function by modifying Linux's existing `do_fork` function such that it took one more argument: a `task_struct` pointer to the process to be forked. Soon we found that this approach would require us to rewrite many existing kernel functions. First, they all needed to take this one extra argument. Second, they all needed to be modified to operate on data structures of the specified process instead of those of the caller process. To avoid this tedious work, we designed an in-kernel fork mechanism that allows us to use the existing `do_fork` function with only slight modifications.

Figure 5 illustrates our in-kernel fork design. To fork process *P*, Pulse first switches *P* in (thus switching itself out), but to a predefined function. Within this function, Process *P* calls `do_fork` to create a copy of itself, *P'*. Finally, it switches the Pulse process back in, which then resumes execution from where it was left off. Similar to ordinary processes, the speculative process *P'* participates in Linux's normal scheduling.

To implement this design, we wrote a simple context switch function, similar to Linux's context switch function. To fork a process *P*, Pulse calls our context switch function, which switches process *P* in by saving the memory and register state of the Pulse process and loading the state of process *P*. With a normal context switch, process *P* would then resume its execution from the

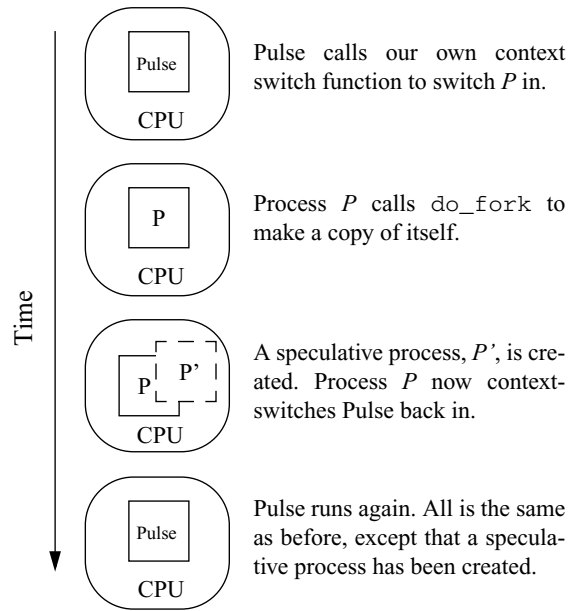


Figure 5: Illustration of in-kernel fork.

point where it was suspended previously. However, in our context switch function, we force it to enter an `in_kernel_fork` function that we added to the kernel. Within this function, process *P* calls `do_fork` to create a speculative process *P'*. It then calls our context switch function to switch the Pulse process in and switch itself out such that it goes back to sleep again.

Certain fields, such as the instruction and stack pointers, in the `task_struct` of process *P* may be changed by the `do_fork` call. Thus, after being switched back in, Pulse restores these fields. Our goal is to ensure that the parent process stays exactly the same before and after the creation of a speculative (child) process. Thus, we modify `do_fork` such that it does not link a speculative process to the children list of its parent. The speculative process does keep a pointer back to its parent, because it needs to know to which sleeping process it corresponds. To allow the speculative process to exit independently of its parent, we reset its parent to be the init process in the `do_exit` function. In `do_fork`, we also initialize the speculative process such that it does not send any signal to its parent when it exits. As such, the speculative process is completely independent of its parent and does not affect the parent in any way.

Starting speculative processes. After a speculative process is created, it participates in normal process scheduling. The normal fork semantics would make the speculative process resume the code in which the parent was suspended. Thus, the speculative process would resume the system call that blocked the parent process previously. However, allowing the speculative process to

execute in the kernel freely may cause changes to important kernel structures (e.g., the semaphore protecting a pipe's inode). Such changes should never occur because they could affect the normal execution of other processes, violating the safety property. To solve this problem, we force all speculative processes to execute a common function, `ret_from_spec_fork`, when they are scheduled to run the first time after creation. This is achieved by setting the initial program counter (EIP) value of each speculative process to be the address of this common function when the process is created.

In `ret_from_spec_fork`, a speculative process first creates the event awaited by its parent as follows. The speculative process checks what events its parent is waiting for by looking up the resource and condition lists maintained for the parent. For example, if the parent is blocked in a `futex` system call, then it must be waiting for the data at a user-space address (*resource*) to become equal to or greater than (*op*) a certain value (*val*). If *op* is *equal-to*, the speculative process writes *val* to the address identified by *resource* within its own address space. If *op* is *greater-than*, it writes *val* + 1 to that address. In fact, for the latter case, any value greater than *val* would be fine since the parent is not waiting for a specific value. However, it is possible that different values may have different meanings. For example, a semaphore's value often represents how many processes are allowed to enter a critical section simultaneously. Thus, our choice of setting the value to *val* + 1 is only heuristic; the parent process may indeed expect a different value at the given address, although it did not explicitly say so when it invoked the `futex` system call.

If the parent process is blocked in a pipe write or `poll` system call, the speculative process is not allowed to create the events awaited by the parent. This is because creating the events requires doing I/O on a pipe, which can affect normal processes that access the pipe.

Regardless of whether the events are created or not, the `ret_from_spec_fork` function then forces the speculative process to exit the system call in which its parent is blocked. This is done by jumping to Linux's `syscall_exit` routine and returning the value that represents success for the corresponding system call. Thus, after returning to the user-level code, the application program will have the illusion that the blocking system call has returned successfully. The speculative process then runs ahead in the program.

Recording events. During execution, a speculative process records all of the events that can awaken a sleeping process. Since our implementation considers only three blocking system calls, we only modify the counterpart system calls of these three calls. Similar to recording

events for which a sleeping process waits, we record the events that a speculative process produces as resource and condition lists too. For the system calls we modify, these lists happen to both have only one element.

A process blocked in the `futex` system call can unblock only if another process calls `futex` with a `FUTEX_WAKE` argument. Thus the `futex` system call is the counterpart of itself. We add code in `futex` such that, when called by a speculative process to perform a wakeup, it records this event. We modify the corresponding NPTL library functions, such as `mutex_unlock` (`pthread_mutex_unlock`) and `sem_post`, such that they pass the necessary information to allow `futex` to fill in the values for the *resource*, *op*, and *val* fields, which characterize the wakeup event produced by the speculative process. For example, if the speculative process invokes `futex` from `pthread_mutex_unlock`, it records that this process is producing an event, that is, the memory location of the mutex (*resource*) now obtains a new value (*val*). The *op* field is set to be the same as what is used for the counterpart system call (in this case, *equal-to*).

A process blocked in a pipe write system call can be unblocked if another process reads the pipe. Thus we add code in the `read` system call to record the read event. When called by a speculative process to read a pipe, the `read` system call records *resource* to be the address of the pipe's inode structure, *op* to be the same as what is used for the corresponding blocking pipe write call, i.e., *and*, and *val* to be `POLLOUT | POLLWRNORM`, which are exactly the bit masks for which the blocking write call is waiting.

For the `poll` system call, we modify two counterpart system calls: `write` and `writew`. For both of them, we record *resource* to be the address of the pipe's inode structure, *op* to be *and*, and *val* to be `POLLIN | POLLRDNORM`, which are kernel-defined macros representing that the pipe has data available to read.

To ensure safety, all of these system calls return immediately with a success code after they record the produced events. Thus, a speculative process calling these system calls does not really perform the wakeup and read/write operations that these calls normally do. Similar to Fraser and Chang [6], we modify all potentially unsafe system calls such that a speculative process returns immediately when entering these calls. In this way, any state change made by a speculative process is contained within itself, not affecting any other process.

Constructing producer edges. A speculative process terminates according to the conditions in Section 3.3. We limit the lifetime of a speculative process to be less

than one second. After all speculative processes terminate, Pulse matches their produced events against those awaited by the sleeping processes. If two events have the same *resource* and *op* values, then Pulse applies the operation represented by *op* on the *val* fields of the two events. For example, if *op* is *and*, then Pulse does a logical *and* on the two *val* fields. A result of true indicates that the speculative process produces an event for which the sleeping process is waiting. Thus Pulse constructs a producer edge from the node that represents the event to the node that represents the parent process of the speculative process.

4.4 Summary

The major components of our implementation are the in-kernel fork and modification of the blocking system calls and their counterpart calls. The in-kernel fork implementation is the most involved part in our coding; however, our code is small and highly efficient, consisting of only 94 lines of C code, 47 lines of inline assembly, and 7 lines of assembly. Modifying the system calls is easy since it simply involves identifying the corresponding resources and conditions, and recording them in a per-process structure (~160 bytes). For this paper, we have modified only three blocking system calls and their counterpart calls. The same methodology, however, can be easily applied to modify other system calls.

5 Evaluation

We apply Pulse against deadlocked solutions to the classical dining-philosophers and smokers problems, and a well-known deadlock scenario in the Apache web server version 2.0.49. This section describes how Pulse works for these different deadlock cases and evaluates its overhead. The evaluation is performed on an IBM xSeries 445 eServer with eight Intel Xeon 2.8 GHz processors and 32 GB memory. We configure Pulse to transition from nap mode to monitor mode with a default value of every five minutes, unless otherwise mentioned. We set the event buffers of speculative processes to store at most ten events, and every speculative process exists in the system for at most one second. Our results show that Pulse can detect deadlocks caused by incorrect ordering of lock acquisitions, as well as deadlocks involving synchronization semaphores and pipes, which, to the best of our knowledge, no existing tools can detect. Furthermore, Pulse generates no false positives and negatives of deadlock throughout our experiments.

```
while (1) {
    think();
    lock(fork[i]);           // take left fork
    lock(fork[(i + 1) % 5]); // take right fork
    eat();
    unlock(fork[i]);         // put left fork
    unlock(fork[(i + 1) % 5]); // put right fork
}
```

Figure 6: The code of philosopher *i*.

5.1 The dining-philosophers problem

Figure 6 shows one incorrect solution to the dining-philosophers problem. We implement the locks as NPTL mutexes. The lock routine is implemented using the `pthread_mutex_lock` function and unlock using `pthread_mutex_unlock`. In Figure 6, suppose that all five philosophers take their left forks simultaneously. Then they will all block on their right forks, and there will be a deadlock. We choose this problem as an example of deadlock caused by incorrect use of mutual exclusion locks. All existing dynamic and static tools target this type of deadlock.

When Pulse enters the detection mode, it identifies that each philosopher process is waiting for an address (corresponding to its right fork) to contain a value zero (corresponding to the release of the right fork). It then forks a speculative process for each philosopher. The speculative processes first unlock their right forks within their own address spaces, and then execute ahead. During execution, they discover that they produce the unlock events that could unblock their left neighbors. Finally, Pulse constructs the resource graph shown in Figure 7. This graph contains a cycle, indicating the existence of a deadlock.

5.2 The smokers problem

The smokers problem is a classic example of using semaphores for synchronization, instead of mutual exclusion. The static deadlock detection tool RacerX [5] specifically ignores checking deadlocks involving such semaphores. With speculative execution, Pulse is able to detect such deadlocks.

Figure 8 shows a solution to the smokers problem that can deadlock. The processes share four binary semaphores: tobacco, paper, matches, and order. The semaphores are implemented using the NPTL library. A P operation is implemented using the `sem_wait` func-

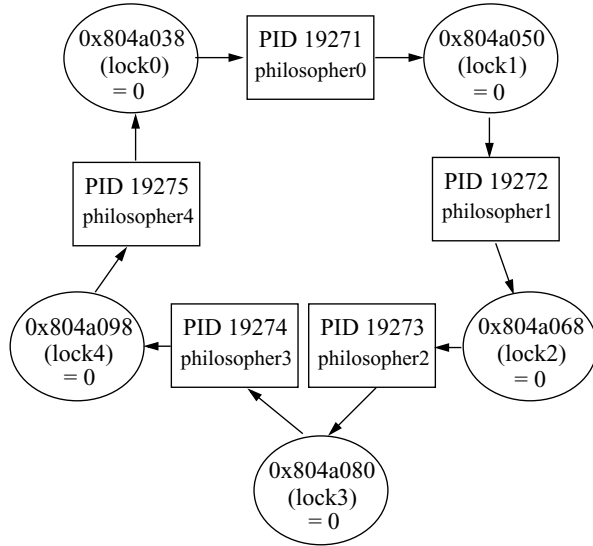


Figure 7: Resource graph for the dining-philosophers problem. The hex numbers are virtual addresses corresponding to the locks.

smoker 1	smoker 2	smoker 3
while (1) { P(tobacco) P(paper) // block V(order) }	while (1) { P(paper) // block P(matches) V(order) }	while (1) { P(matches) P(tobacco) // block V(order) }
agent while (1) { P(order) // block V(one of tobacco, paper, matches at random) V(one of the three at random but not above) }		

Figure 8: A deadlocked solution to the smokers problem.

tion, and V is implemented using `sem_post`. In Figure 8, suppose that the agent releases tobacco and matches, smoker 1 grabs the tobacco, and smoker 3 grabs the matches. Then smoker 1 will block in `P(paper)`, waiting for the address corresponding to paper to have a value greater than zero. Similarly, smoker 2 will block in `P(paper)`, smoker 3 in `P(tobacco)`, and the agent in `P(order)`. Thus all processes will deadlock.

By speculatively unblocking smoker 1, Pulse observes an event corresponding to `V(order)`, which matches the event awaited by the agent. Thus Pulse constructs a producer edge from an event node representing `V(order)` to smoker 1's node. Then the speculative process executes `P(tobacco)` again. Tobacco is not available any more, so this speculative process blocks again. For smoker 2, after the speculative process is unblocked from `P(paper)`, it blocks immediately in `P(matches)`, and

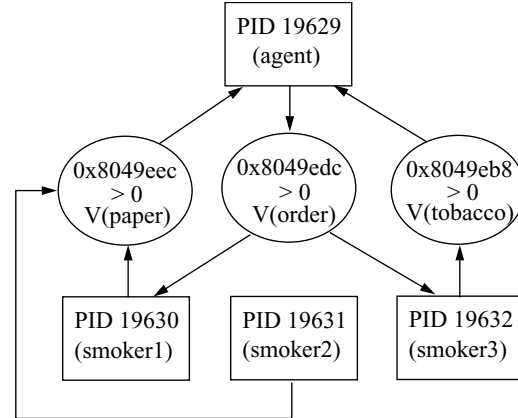


Figure 9: Resource graph for the smoker's problem. The hex numbers are virtual addresses corresponding to the semaphores.

thus it does not produce any event that could unblock another process. So the process node of smoker 2 has no incoming producer edges. Smoker 3 executes similarly to smoker 1.

Suppose that after the agent is speculatively unblocked, it executes `V(tobacco)` and `V(paper)`. Then Pulse will discover producer edges from both the event nodes, corresponding to `V(tobacco)` and `V(paper)`, to the agent. Figure 9 shows the final graph we obtain. This graph contains two cycles, `agent` → `V(order)` → `smoker 1` → `V(paper)` → `agent`, and `agent` → `V(order)` → `smoker 3` → `V(tobacco)` → `agent`, indicating the existence of deadlock.

5.3 Apache web server deadlock

We have reproduced a well-known deadlock situation in Apache 2.0.49 with the prefork Multi-Processing Module (MPM). A full description of this bug (number 22030) can be found in the Apache bug database (<http://nagoya.apache.org/bugzilla/>). This bug is widely referred to as the Apache oversized stderr buffer denial-of-service vulnerability by security companies like Symantec [12]. In the extreme case, this bug can cause an entire web site to stop functioning.

To reproduce the deadlock effect of this bug, we create a Perl CGI script, which first writes 4097 bytes to `stderr` and then some arbitrary data to `stdout`. When a client requests this CGI script from a remote browser, a deadlock happens on the server side. This deadlock involves two processes: the CGI script's process and the `httpd` process that handles the CGI request. The reason for this deadlock is because Apache redirects `stderr` and `stdout` of the CGI script to two pipes. The pipe buffer size in Linux is 4096 bytes; a blocking write to a pipe

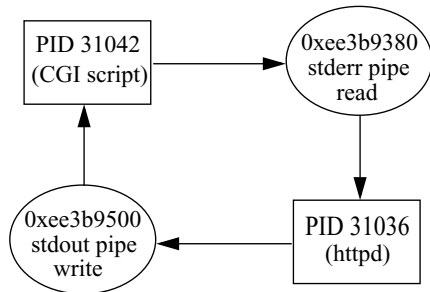


Figure 10: Resource graph for the Apache deadlock. The hex numbers are addresses of the corresponding pipe inode structures. We label the events in words although Pulse encodes them abstractly.

blocks if the write data exceeds 4096 bytes. Thus when the CGI script writes 4097 bytes to stderr, it blocks in the `write` system call, waiting for the httpd process to read data out of the pipe. Meanwhile, the httpd process is blocked in the `poll` system call, waiting for the CGI script to write data to the stdout pipe.

No existing techniques can detect this deadlock; with Pulse, we successfully detect it. Figure 10 shows the resource graph constructed by Pulse. This graph contains a cycle, indicating the existence of deadlock. It is worth noting that Apache has a timeout mechanism that allows it to fail the CGI request after about five minutes, thus breaking the deadlock. However, Apache provides no debugging information for the deadlock; it even has no idea that a deadlock has occurred. When this deadlock is triggered simultaneously from multiple sites, it can effectively become a denial-of-service attack.

5.4 Performance Overhead

We evaluate three aspects of Pulse’s performance overhead: overhead of the modified system calls, overhead of periodic checking, and overhead of deadlock detection using speculative execution.

System call overhead. We write a microbenchmark for each of the three blocking system calls we modify. Our goal is to measure the overhead that our modified system calls introduce to correct, non-deadlocked applications. Since the additional code in our modified system calls that counterpart the blocking calls is executed only by speculative processes, we do not measure the overhead of these counterpart calls.

Our microbenchmark for the `futex` system call repeatedly invokes `futex` for a total of one million times. The arguments passed to `futex` are chosen such that the microbenchmark executes the new code we add, but it does not block (thus allowing the microbenchmark to repeatedly invoke the system call). We run the

microbenchmark using the modified and unmodified `futex` call, and compare the average time taken per call. We run similar microbenchmarks for `write` and `poll`. Compared to the unmodified versions, the slowdowns of our modified system calls are: 0.2% for `futex`, 0.9% for `write`, and 1% for `poll`. These slowdowns are small, and most importantly, they occur mostly when a process is about to block.

Periodic checking overhead. We evaluate the performance overhead that Pulse introduces to correct, non-deadlocked applications. This overhead comes from Pulse’s periodic activities of transitioning from nap to monitor mode and checking if it needs to enter the detection mode. For correct applications, Pulse does not enter the detection mode—after the periodic checking, it returns back to the nap mode. Our experiments show that, for a system with 100 threads, Pulse takes 0.29 seconds to transition from nap to monitor mode, scan all the threads to determine if it needs to enter the detection mode, and finally transition back to the nap mode. This time stays almost constant as we create more threads in the system, and only increases to 0.32 seconds when the system has a total of 2000 threads.

To measure how much performance overhead Pulse introduces to normal applications over time, we run Apache Bench from a remote machine to stress test our server running Apache 2.0.49. We set Apache Bench to perform a total of five million HTTP requests and send 1000 simultaneous requests at any time (thus keeping the server busy). The entire test takes about 30 minutes to complete. During this period, without Pulse running, Apache Bench obtains throughput of 2689 requests per second. We then run Pulse in the background, and set it to periodically transition from nap to monitor mode every one minute. Apache Bench now obtains throughput of 2684 requests per second, which is almost the same as the throughput obtained without Pulse running. These results show that Pulse has negligible impact on the performance of applications that do not deadlock.

Deadlock detection overhead. We measure the time Pulse takes to detect a deadlock, which is the duration from the time Pulse enters the detection mode to the time it finishes the detection and prints out the results. We obtain the following results: it takes Pulse 2.1 seconds to detect the deadlock in the dining-philosophers problem, 1.7 seconds in the smokers problem, and 1.5 seconds in the Apache web server. We also run these three benchmarks together such that they all deadlock at the same time. We see that Pulse can construct a general resource graph that has three subgraphs, each corresponding to one of the deadlock scenarios, and the entire detection only takes three seconds.

6 Conclusion

Deadlock can occur in any concurrent system and is often difficult to debug. Existing deadlock detection techniques are either impractical for large software systems or over-simplified in their assumptions about deadlock-sensitive resources. In this paper, we propose Pulse, a novel operating system mechanism that dynamically detects deadlock in user applications.

Pulse runs as a system daemon. Periodically, it identifies long sleeping processes and the events they are waiting for. For each of these processes, Pulse forks a speculative process, which executes ahead in its parent's program. Speculative execution enables Pulse to discover dependences among the sleeping processes. Based on this information, it constructs a general resource graph. If the graph contains cycles, Pulse outputs that deadlock exists. It also prints out the entire graph to help application developers identify causes of the deadlock.

Our evaluation demonstrates that Pulse can detect various types of deadlock, including those involving consumable resources, which no existing tool can detect. Our results show that Pulse can detect deadlock quickly and that it introduces little performance overhead to normal applications that do not deadlock. For application developers, Pulse can be viewed as another tool to add to the deadlock detection toolbox. When Pulse and the existing tools are used together, they can provide the best coverage of deadlocks.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions on the early draft of this paper. We thank David Becker and Jaidev Patwardhan for their help with the experiments. This work is supported in part by the US National Science Foundation (CCR-0312561, EIA-9972879, CCR-0204367, CCR-0208920, and CCR-0309164), Intel, IBM, Microsoft and the Duke University Graduate School. Sorin is supported by a Warren Faculty Scholarship.

References

- [1] Ruediger R. Asche. Putting DLDETECT to Work. MSDN Library Technical Articles, Microsoft Corporation, January 1994.
- [2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, pages 164–177, October 2003.
- [3] Fay Chang and Garth A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pages 1–14, February 1999.
- [4] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 1999.
- [5] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlock. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, pages 237–252, October 2003.
- [6] Keir Fraser and Fay Chang. Operating System I/O Speculation: How Two Invocations Are Faster Than One. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 325–338, June 2003.
- [7] Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of The 24th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 174–186, January 1997.
- [8] Klaus Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *Proceedings of the 7th SPIN Workshop*, pages 245–264, August 2000.
- [9] Richard C. Holt. Some Deadlock Properties of Computer Systems. *ACM Computing Surveys*, 4(3):179–196, September 1972.
- [10] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [11] Mukesh Singhal. Deadlock Detection in Distributed Systems. *IEEE Computer*, 22(11):37–48, November 1989.
- [12] Symantec. *Symantec NetRecon™ 3.6 Security Update 6 Release Notes*. Symantec Corporation, August 2003.
- [13] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Journal of Automated Software Engineering*, 10(2):203–232, April 2003.
- [14] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 181–194, December 2002.

Surviving Internet Catastrophes

Flavio Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo and Geoffrey M. Voelker

Department of Computer Science and Engineering

University of California, San Diego

{flavio,rbhagwan,ahevia,marzullo,voelker}@cs.ucsd.edu

Abstract

In this paper, we propose a new approach for designing distributed systems to survive Internet catastrophes called informed replication, and demonstrate this approach with the design and evaluation of a cooperative backup system called the Phoenix Recovery Service. Informed replication uses a model of correlated failures to exploit software diversity. The key observation that makes our approach both feasible and practical is that Internet catastrophes result from shared vulnerabilities. By replicating a system service on hosts that do not have the same vulnerabilities, an Internet pathogen that exploits a vulnerability is unlikely to cause all replicas to fail. To characterize software diversity in an Internet setting, we measure the software diversity of host operating systems and network services in a large organization. We then use insights from our measurement study to develop and evaluate heuristics for computing replica sets that have a number of attractive features. Our heuristics provide excellent reliability guarantees, result in low degree of replication, limit the storage burden on each host in the system, and lend themselves to a fully distributed implementation. We then present the design and prototype implementation of Phoenix, and evaluate it on the PlanetLab testbed.

1 Introduction

The Internet today is highly vulnerable to Internet epidemics: events in which a particularly virulent Internet pathogen, such as a worm or email virus, compromises a large number of hosts. Starting with the Code Red worm in 2001, which infected over 360,000 hosts in 14 hours [27], such pathogens have become increasingly virulent in terms of speed, extent, and sophistication. Sapphire scanned most IP addresses in less than 10 minutes [25], Nimda reportedly infected millions of hosts, and Witty exploited vulnerabilities in firewall software explicitly designed to defend hosts from such pathogens [26]. We call such epidemics *Internet catastrophes* because they result in extensive wide-spread damage costing billions of dollars [27]. Such damage ranges from overwhelming networks with epidemic traffic [25, 27], to providing zombies for spam relays [30] and denial of service attacks [35], to deleting disk blocks [26]. Given the current ease with which such pathogens can be created and launched,

further Internet catastrophes are inevitable in the near future.

Defending hosts and the systems that run on them is therefore a critical problem, and one that has received considerable attention recently. Approaches to defend against Internet pathogens generally fall into three categories. Prevention reduces the size of the vulnerable host population [38, 41, 42]. Treatment reduces the rate of infection [9, 33]. Finally, containment techniques block infectious communication and reduce the contact rate of a spreading pathogen [28, 44, 45].

Such approaches can mitigate the impact of an Internet catastrophe, reducing the number of vulnerable and compromised hosts. However, they are unlikely to protect all vulnerable hosts or entirely prevent future epidemics and risk of catastrophes. For example, fast-scanning worms like Sapphire can quickly probe most hosts on the Internet, making it challenging for worm defenses to detect and react to them at Internet scale [28]. The recent Witty worm embodies a so-called *zero-day worm*, exploiting a vulnerability soon after patches were announced. Such pathogens make it increasingly difficult for organizations to patch vulnerabilities before a catastrophe occurs. As a result, we argue that defenses are necessary, but not sufficient, for fully protecting distributed systems and data on Internet hosts from catastrophes.

In this paper, we propose a new approach for designing distributed systems to survive Internet catastrophes called informed replication. The key observation that makes informed replication both feasible and practical is that Internet epidemics exploit shared vulnerabilities. By replicating a system service on hosts that do not have the same vulnerabilities, a pathogen that exploits one or more vulnerabilities cannot cause all replicas to fail. For example, to prevent a distributed system from failing due to a pathogen that exploits vulnerabilities in Web servers, the system can place replicas on hosts running different Web server software.

The software of every system inherently is a shared vulnerability that represents a risk to using the system, and systems designed to use informed replication are no different. Substantial effort has gone into making systems themselves more secure, and our design approach can certainly benefit from this effort. However, with the dramatic rise of worm epidemics, such systems are now increasingly at risk to large-scale failures due to vulnerabilities in *unrelated* software running on the host. Informed replication reduces this new source of risk.

This paper makes four contributions. First, we develop a system model using the *core* abstraction [15] to represent failure correlation in distributed systems. A core is a reliable minimal subset of components such that the probability of having all hosts in a core failing is negligible. To reason about the correlation of failures among hosts, we associate *attributes* with hosts. Attributes represent characteristics of the host that can make it prone to failure, such as its operating system and network services. Since hosts often have many characteristics that make it vulnerable to failure, we group host attributes together into *configurations* to represent the set of vulnerabilities for a host. A system can use the configurations of all hosts in the system to determine how many replicas are needed, and on which hosts those replicas should be placed, to survive a worm epidemic.

Second, the efficiency of informed replication fundamentally depends upon the degree of software diversity among the hosts in the system, as more homogeneous host populations result in a larger storage burden for particular hosts. To evaluate the degree of software heterogeneity found in an Internet setting, we measure and characterize the diversity of the operating systems and network services of hosts in the UCSD network. The operating system is important because it is the primary attribute differentiating hosts, and network services represent the targets for exploit by worms. The results of this study indicate that such networks have sufficient diversity to make informed replication feasible.

Third, we develop heuristics for computing cores that have a number of attractive features. They provide excellent reliability guarantees, ensuring that user data survives attacks of single- and double-exploit pathogens with probability greater than 0.99. They have low overhead, requiring fewer than 3 copies to cope with single-exploit pathogens, and fewer than 5 copies to cope with double-exploit pathogens. They bound the number of replica copies stored by any host, limiting the storage burden on any single host. Finally, the heuristics lend themselves to a fully distributed implementation for scalability. Any host can determine its replica set (its core) by contacting a constant number of other hosts in the system, independent of system size.

Finally, to demonstrate the feasibility and utility of our approach, we apply informed replication to the design and implementation of Phoenix. Phoenix is a cooperative, distributed remote backup system that protects stored data against Internet catastrophes that cause data loss [26]. The usage model of Phoenix is straightforward: users specify an amount F of bytes of their disk space for management by the system, and the system protects a proportional amount F/k of their data using storage provided by other hosts, for some value of k . We implement Phoenix as a service layered on the Pastry DHT [32] in the Macedon framework [31], and evaluate its ability to survive emulated catastrophes on the PlanetLab testbed.

The rest of this paper is organized as follows. Section 2 dis-

cusses related work. Section 3 describes our system model for representing correlated failures. Section 4 describes our measurement study of the software diversity of hosts in a large network, and Section 5 describes and evaluates heuristics for computing cores. Section 6 describes the design and implementation of Phoenix, and Section 7 describes the evaluation of Phoenix. Finally, Section 8 concludes.

2 Related work

Most distributed systems are not designed such that failures are independent, and there has been recent interest in protocols for systems where failures are correlated. Quorum-based protocols, which implement replicated update by reading and writing overlapping subsets of replicas, are easily adapted to correlated failures. A model of dependent failures was introduced for Byzantine-tolerant quorum systems [23]. This model, called a *fail-prone system*, is a dual representation of the model (*cores*) that we use here. Our model was developed as part of a study of lower bounds and optimal protocols for Consensus in environments where failures can be correlated [15].

The ability of Internet pathogens to spread through a vulnerable host population on the network fundamentally depends on three properties of the network: the number of susceptible hosts that could be infected, the number of infected hosts actively spreading the pathogen, and the contact rate at which the pathogen spreads. Various approaches have been developed for defending against such epidemics that address each of these properties.

Prevention techniques, such as patching [24, 38, 42] and overflow guarding [7, 41], prevent pathogens from exploiting vulnerabilities, thereby reducing the size of the vulnerable host population and limiting the extent of a worm outbreak. However, these approaches have the traditional limitations of ensuring soundness and completeness, or leave windows of vulnerability due to the time required to develop, test, and deploy.

Treatment techniques, such as disinfection [6, 9] and vaccination [33], remove software vulnerabilities after they have been exploited and reduce the rate of infection as hosts are treated. However, such techniques are reactive in nature and hosts still become infected.

Containment techniques, such as throttling [21, 44] and filtering [28, 39], block infectious communication between infected and uninfected hosts, thereby reducing or potentially halting the contact rate of a spreading pathogen. The efficacy of reactive containment fundamentally depends upon the ability to quickly detect a new pathogen [19, 29, 37, 46], characterize it to create filters specific to infectious traffic [10, 16, 17, 34], and deploy such filters in the network [22, 40]. Unfortunately, containment at Internet scales is challenging, requiring short reaction times and extensive

deployment [28, 45]. Again, since containment is inherently reactive, some hosts always become infected.

Various approaches take advantage of software heterogeneity to make systems fault-tolerant. N-version programming uses different implementations of the same service to prevent correlated failures across implementations. Castro's Byzantine fault tolerant NFS service (BFS) is one such example [4] and provides excellent fault-tolerant guarantees, but requires multiple implementations of every service. Scrambling the layout and execution of code can introduce heterogeneity into deployed software [1]. However, such approaches can make debugging, troubleshooting, and maintaining software considerably more challenging. In contrast, our approach takes advantage of existing software diversity.

Lastly, Phoenix is just one of many proposed cooperative systems for providing archival and backup services. For example, Intermemory [5] and Oceanstore [18] enable stored data to persist indefinitely on servers distributed across the Internet. As with Phoenix, Oceanstore proposes mechanisms to cope with correlated failures [43]. The approach, however, is reactive and does not enable recovery after Internet catastrophes. With Pastiche [8], pStore [2], and CIBS [20], users relinquish a fraction of their computing resources to collectively create a backup service. However, these systems target localized failures simply by storing replicas offsite. Such systems provide similar functionality as Phoenix, but are not designed to survive wide-spread correlated failures of Internet catastrophes. Finally, Glacier is a system specifically designed to survive highly correlated failures like Internet catastrophes [11]. In contrast to Phoenix, Glacier assumes a very weak failure model and instead copes with catastrophic failures via massive replication. Phoenix relies upon a stronger failure model, but replication in Phoenix is modest in comparison.

3 System model

As a first step toward the development of a technique to cope with Internet catastrophes, in this section we describe our system model for representing and reasoning about correlated failures, and discuss the granularity at which we represent software diversity.

3.1 Representing correlated failures

Consider a system composed of a set \mathcal{H} of hosts each of which is capable of holding certain objects. These hosts can fail (for example, by crashing) and, to keep these objects available, they need to be replicated. A simple replication strategy is to determine the maximum number t of hosts that can fail at any time, and then maintain more than t replicas of each object.

However, using more than t replicas may lead to excessive replication when host failures are correlated. As a simple ex-

ample, consider three hosts $\{h_1, h_2, h_3\}$ where the failures of h_1 and h_2 are correlated while h_3 fails independent of the other hosts. If h_1 fails, then the probability of h_2 failing is high. As a result, one might set $t = 2$ and thereby require $t + 1 = 3$ replicas. However, if we place replicas on h_1 and h_3 , the object's availability may be acceptably high with just two replicas.

To better address issues of optimal replication in the face of correlated failures, we have defined an abstraction that we call a *core* [15]. A core is a minimal set of hosts such that, in any execution, at least one host in the core does not fail. In the above example, both $\{h_1, h_3\}$ and $\{h_2, h_3\}$ are cores. $\{h_1, h_2\}$ would not be a core since the probability of both failing is too high and $\{h_1, h_2, h_3\}$ would not be a core since it is not minimal. Using this terminology, a central problem of informed replication is the identification of cores based on the correlation of failures.

An Internet catastrophe causes hosts to fail in a correlated manner because all hosts running the targeted software are vulnerable. Operating systems and Web servers are examples of software commonly exploited by Internet pathogens [27, 36]. Hence we characterize a host's vulnerabilities by the software they run. We associate with each host a set of *attributes*, where each attribute is a canonical name of a software package or system that the host runs; in Section 3.2 below, we discuss the tradeoffs of representing software packages at different granularities. We call the combined representation of all attributes of a host the *configuration* of the host. An example of a configuration is $\{\text{Windows}, \text{IIS}, \text{IE}\}$, where *Windows* is a canonical name for an operating system, *IIS* for a Web server package, and *IE* for a Web browser. Agreeing on canonical names for attribute values is essential to ensure that dependencies of host failures are appropriately captured.

An Internet pathogen can be characterized by the set of attributes A that it targets. Any host that has none of the attributes in A is not susceptible to the pathogen. A core is a minimal set C of hosts such that, for each pathogen, there is a host h in C that is not susceptible to the pathogen. Internet pathogens often target a single (possibly cross-platform) vulnerability, and the ones that target multiple vulnerabilities target the same operating system. Assuming that any attribute is susceptible to attack, we can re-define a core using attributes: a core is a minimal set C of processes such that no attribute is common to all hosts in C . In Section 5.4, we relax this assumption and show how to extend our results to tolerate pathogens that can exploit multiple vulnerabilities.

To illustrate these concepts, consider the system described in Example 3.1. In this system, hosts are characterized by six attributes which we classify for clarity into operating system, Web server, and Web browser.

H_1 and H_2 comprise what we call an *orthogonal core*, which is a core composed of hosts that have disjoint configurations. Given our assumption that Internet pathogens

target only one vulnerability or multiple vulnerabilities on one platform, an orthogonal core will contain two hosts. $\{H_1, H_3, H_4\}$ is also a core because there is no attribute present in all hosts, and it is minimal.

Example 3.1

Attributes: *Operating System* = {Unix, Windows};
 Web Server = {Apache, IIS};
 Web Browser = {IE, Netscape}.

Hosts: $H_1 = \{\text{Unix, Apache, Netscape}\};$
 $H_2 = \{\text{Windows, IIS, IE}\};$
 $H_3 = \{\text{Windows, IIS, Netscape}\};$
 $H_4 = \{\text{Windows, Apache, IE}\}.$

Cores = $\{\{H_1, H_2\}, \{H_1, H_3, H_4\}\}.$

The smaller core $\{H_1, H_2\}$ might appear to be the better choice since it requires less replication. Choosing the smallest core, however, can have an adverse effect on individual hosts if many hosts use this core for placing replicas. To represent this effect, we define *load* to be the amount of storage a host provides to other hosts. In environments where some configurations are rare, hosts with the rare configurations may occur in a large percentage of the smallest cores. Thus, hosts with rare configurations may have a significantly higher load than the other hosts. Indeed, having a rare configuration can increase a host's load even if the smallest core is not selected. For example, in Example 3.1, H_1 is the only host that has a flavor of Unix as its operating system. Consequently, H_1 is present in both cores.

To make our argument more concrete, consider the worms in Table 1, which are well-known worms unleashed in the past few years. For each worm, given two hosts with one not running Windows or not running a specific server such as a Web server or a database, at least one survives the attack. With even a very modest amount of heterogeneity, our method of constructing cores includes such pairs of hosts.

3.2 Attribute granularity

Attributes can represent software diversity at many different granularities. The choice of attribute granularity balances resilience to pathogens, flexibility for placing replicas, and degree of replication. An example of the coarsest representation is for a host to have a configuration comprising a single attribute for the generic class of operating system, e.g., "Windows", "Unix", etc. This single attribute represents the potential vulnerabilities of all versions of software running on all versions of the same class of operating system. As a result, replicas would always be placed on hosts with different operating systems. A less coarse representation is to have attributes for the operating system as well as all network services running on the host. This representation yields more freedom for placing replicas. For example, we can place replicas on hosts with the same class of operating system if

Worm	Form of infection (Service)	Platform
Code Red	port 80/http (MS IIS)	Windows
Nimda	multiple: email; Trojan horse versions using open network shares (SMB: ports 137–139 and 445); port 80/HTTP (MS IIS); Code Red backdoors	Windows
Sapphire	port 1434/udp (MS SQL, MSDE)	Windows
Sasser	port 445/tcp (LSASS)	Windows
Witty	port 4000/udp (BlackICE)	Windows

Table 1: Recent well-known pathogens.

they run different services. The core $\{H_1, H_3, H_4\}$ in Example 3.1 is an example of this situation since H_3 and H_4 both run Windows. More fine-grained representations can have attributes for different versions of operating systems and applications. For example, we can represent the various releases of Windows, such as "Windows 2000" and "Windows XP", or even versions such as "NT 4.0sp4" as attributes. Such fine-grained attributes provide considerable flexibility in placing replicas. For example, we can place a replica on an NT host and an XP host to protect against worms such as Code Red that exploit an NT service but not an XP service. But doing so greatly increases the cost and complexity of collecting and representing host attributes, as well as computing cores to determine replica sets.

Our initial work [14] suggested that informed replication can be effective with relatively coarse-grained attributes for representing software diversity. As a result, we use attributes that represent just the class of operating system and network services on hosts in the system, and not their specific versions. In subsequent sections, we show that, when representing diversity at this granularity, hosts in an enterprise-scale network have substantial and sufficient software diversity for efficiently supporting informed replication. Our experience suggests that, although we can represent software diversity at finer attribute granularities such as specific software versions, there is not a compelling need to do so.

4 Host diversity

With informed replication, the difficulty of identifying cores and the resulting storage load depend on the actual distribution of attributes among a set of hosts. To better understand these two issues, we measured the software diversity of a large set of hosts at UCSD. In this section, we first describe the methodology we used, and discuss the biases and limitations our methodology imposes. We then characterize the operating system and network service attributes found on the hosts, as well as the host configurations formed by those attributes.

4.1 Methodology

On our behalf, UCSD Network Operations used the *Nmap* tool [12] to scan IP address blocks owned by UCSD to deter-

mine the host type, operating system, and network services running on the host. Nmap uses various scanning techniques to classify devices connected to the network. To determine operating systems, Nmap interacts with the TCP/IP stack on the host using various packet sequences or packet contents that produce known behaviors associated with specific operating system TCP/IP implementations. To determine the network services running on hosts, Nmap scans the host port space to identify all open TCP and UDP ports on the host. We anonymized host IP addresses prior to processing.

Due to administrative constraints collecting data, we obtained the operating system and port data at different times. We had a port trace collected between December 19–22, 2003, and an operating system trace collected between December 29, 2003 and January 7, 2004. The port trace contained 11,963 devices and the operating system trace contained 6,395 devices.

Because we are interested in host data, we first discarded entries for specialized devices such as printers, routers, and switches. We then merged these traces to produce a combined trace of hosts that contained both operating system data and open port data for the same set of hosts. When fingerprinting operating systems, Nmap determines both a class (e.g., Windows) as well as a version (e.g., Windows XP). For added consistency, we discarded host information for those entries that did not have consistent OS class and version info. The result was a data set with operating system and port data for 2,963 general-purpose hosts.

Our data set was constructed using assumptions that introduced biases. First, worms exploit vulnerabilities that are present in network services. We make the assumption that two hosts that have the same open port are running the same network service and thus have the same vulnerability. In fact, two hosts may use a given port to run different services, or even different versions (with different vulnerabilities) of the same service. Second, ignoring hosts that Nmap could not consistently fingerprint could bias the host traces that were used. Third, DHCP-assigned host addresses are reused. Given the time elapsed between the time operating system information was collected and port information was collected, an address in the operating system trace may refer to a different host in the port trace. Further, a host may appear multiple times with different addresses either in the port trace or in the operating system trace. Consequently, we may have combined information from different hosts to represent one host or counted the same host multiple times.

The first assumption can make two hosts appear to share vulnerabilities when in fact they do not, and the second assumption can consistently discard configurations that otherwise contribute to a less skewed distribution of configurations. The third assumption may make the distribution of configurations seem less skewed, but operating system and port counts either remain the same (if hosts do not appear multiple times in the traces) or increase due to repeated configurations.

OS		Port	
Name	Count (%)	Number	Count (%)
Windows	1604 (54.1)	139 (netbios-ssn)	1640 (55.3)
Solaris	301 (10.1)	135 (epmap)	1496 (50.4)
Mac OS X	296 (10.0)	445 (microsoft-ds)	1157 (39.0)
Linux	296 (10.0)	22 (sshd)	910 (30.7)
Mac OS	204 (6.9)	111 (sunrpc)	750 (25.3)
FreeBSD	66 (2.2)	1025 (various)	735 (24.8)
IRIX	60 (2.0)	25 (smtp)	575 (19.4)
HP-UX	32 (1.1)	80 (httpd)	534 (18.0)
BSD/OS	28 (0.9)	21 (ftpd)	528 (17.8)
Tru64 Unix	22 (0.7)	515 (printer)	462 (15.6)

(a)

(b)

Table 2: Top 10 operating systems (a) and ports (b) among the 2,963 general-purpose hosts.

The net effect of our assumptions is to make operating system and port distributions appear to be less diverse than it really is, although it may have the opposite effect on the distribution of configurations.

Another bias arises from the environment we surveyed. A university environment is not necessarily representative of the Internet, or specific subsets of it. We suspect that such an environment is more diverse in terms of software use than other environments, such as the hosts in a corporate environment or in a governmental agency. On the other hand, there are perhaps thousands of universities with a large setting connected to the Internet around the globe, and so the conclusions we draw from our data are undoubtedly not singular.

4.2 Attributes

Together, the hosts in our study have 2,569 attributes representing operating systems and open ports. Table 2 shows the ten most prevalent operating systems and open ports identified on the general purpose hosts. Table 2.a shows the number and percentage of hosts running the named operating systems. As expected, Windows is the most prevalent OS (54% of general purpose hosts). Individually, Unix variants vary in prevalence (0.03–10%), but collectively they comprise a substantial fraction of the hosts (38%).

Table 2.b shows the most prevalent open ports on the hosts and the network services typically associated with those port numbers. These ports correspond to services running on hosts, and represent the points of vulnerability for hosts. On average, each host had seven ports open. However, the number of ports per host varied considerably, with 170 hosts only having one port open while one host (running a firewall software) had 180 ports open. Windows services dominate the network services running on hosts, with netbios-ssn (55%), epmap (50%), and domain services (39%) topping the list. The most prevalent services typically associated with Unix are sshd (31%) and sunrpc (25%). Web servers on port 80 are roughly as prevalent as ftp (18%).

These results show that the software diversity is signifi-

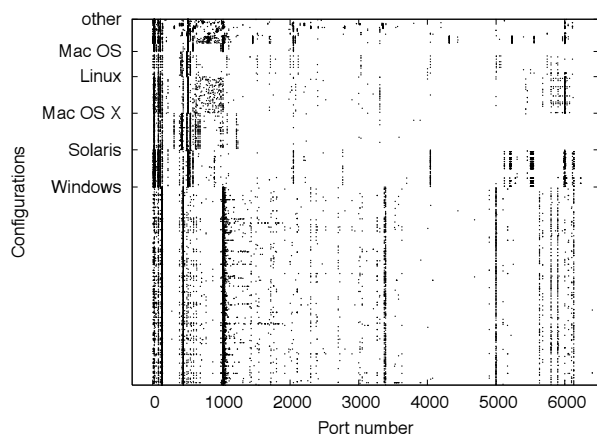


Figure 1: Visualization of UCSD configurations.

cantly skewed. Most hosts have open ports that are shared by many other hosts (Table 2.b lists specific examples). However, most attributes are found on few hosts, i.e., most open ports are open on only a few hosts. From our traces, we observe that the first 20 most prevalent attributes are found on 10% or more of hosts, but the remaining attributes are found on fewer hosts.

These results are encouraging for the process of finding cores. Having many attributes that are not widely shared makes it easier to find replicas that cover each other's attributes, preventing a correlated failure from affecting all replicas. We examine this issue next.

4.3 Configurations

Each host has multiple attributes comprised of its operating system and network services, and together these attributes determine its configuration. The distribution of configurations among the hosts in the system determines the difficulty of finding core replica sets. The more configurations shared by hosts, the more challenging it is to find small cores.

Figure 1 is a qualitative visualization of the space of host configurations. It shows a scatter plot of the host configurations among the UCSD hosts in our study. The x-axis is the port number space from 0–6500, and the y-axis covers the entire set of 2,963 host configurations grouped by operating system family. A dot corresponds to an open port on a host, and each horizontal slice of the scatter plot corresponds to the configuration of open ports for a given host. We sort groups in decreasing size according to the operating systems listed in Table 2: Windows hosts start at the bottom, then Solaris, Mac OS X, etc. Note that we have truncated the port space in the graph; hosts had open ports above 6500, but showing these ports did not add any additional insight and obscured patterns at lower, more prevalent port numbers.

Figure 1 shows a number of interesting features of the configuration space. The marked vertical bands within each group indicate, as one would expect, strong correlations of

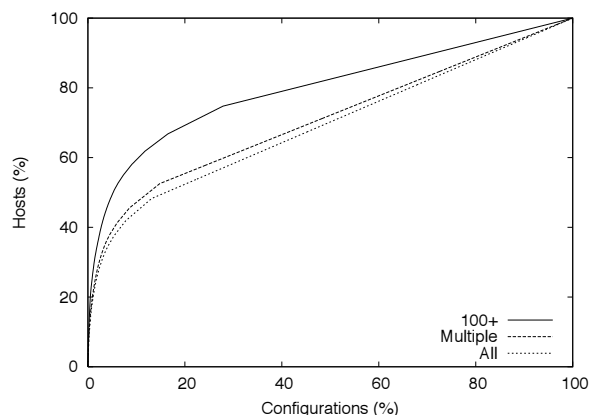


Figure 2: Distribution of configurations.

network services among hosts running the same general operating system. For example, most Windows hosts run the `epmap` (port 135) and `netbios` (port 139) services, and many Unix hosts run `sshd` (port 22) and `X11` (port 6000). Also, in general, non-Windows hosts tend to have more open ports (8.3 on average) than Windows hosts (6.0 on average). However, the groups of hosts running the same operating system still have substantial diversity within the group. Although each group has strong bands, they also have a scattering of open ports between the bands contributing to diversity within the group. Lastly, there is substantial diversity among the groups. Windows hosts have different sets of open ports than hosts running variants of Unix, and these sets even differ among Unix variants. We take advantage of these characteristics to develop heuristics for determining cores in Section 5.

Figure 2 provides a quantitative evaluation of the diversity of host configurations. It shows the cumulative distribution of configurations across hosts for different classes of port attributes, with configurations on the x-axis sorted by decreasing order of prevalence. A distribution in which all configurations are equally prevalent would be a straight diagonal line. Instead, the results show that the distribution of configurations is skewed, with a majority of hosts accounting for only a small percentage of all configurations. For example, when considering all attributes, 50% of hosts comprise just 20% of configurations. In addition, reducing the number of port attributes considered further skews the distribution. For example, when only considering ports that appear on more than one host, shown by the “Multiple” line, 15% of the configurations represent over 50% of the hosts. And when considering only the port attributes that appear on at least 100 hosts, only 8% of the configurations represent over 50% of the hosts. Skew in the configuration distribution makes it more difficult to find cores for those hosts that share more prevalent configurations with other hosts. In the next section, however, we show that host populations with diversity similar to UCSD are sufficient for efficiently constructing cores that result in a low storage load.

5 Surviving catastrophes

With informed replication, each host h constructs a core $Core(h)$ based on its configuration and the configuration of other hosts.¹ Unfortunately, computing a core of optimal size is NP-hard, as we have shown with a reduction from SET-COVER [13]. Hence, we use heuristics to compute $Core(h)$. In this section, we first discuss a structure for representing advertised configurations that is amenable to heuristics for computing cores. We then describe four heuristics and evaluate via simulation the properties of the cores that they construct. As a basis for our simulations, we use the set of hosts \mathcal{H} obtained from the traces discussed in Section 4.

5.1 Advertised configurations

Our heuristics are different versions of greedy algorithms: a host h repeatedly selects other hosts to include in $Core(h)$ until some condition is met. Hence we chose a representation that makes it easier for a greedy algorithm to find good candidates to include in $Core(h)$. This representation is a three-level hierarchy.

The top level of the hierarchy is the operating system that a host runs, the second level includes the applications that run on that operating system, and the third level are hosts. Each host runs one operating system, and so each host is subordinate to its operating system in the hierarchy (we can represent hosts running multiple virtual machines as multiple virtual hosts in a straightforward manner). Since most applications run predominately on one platform, hosts that run a different operating system than h are likely good candidates for including in $Core(h)$. We call the first level the *containers* and the second level the *sub-containers*. Each sub-container contains a set of hosts. Figure 3 illustrates these abstractions using the configurations of Example 3.1.

More formally, let \mathcal{O} be the set of canonical operating system names and \mathcal{C} be the set of containers. Each host h has an attribute $h.os$ that is the canonical name of the operating system on h . The function $m_c : \mathcal{O} \rightarrow \mathcal{C}$ maps operating system name to container; thus, $m_c(h.os)$ is the container that contains h .

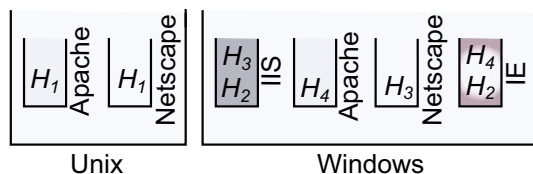


Figure 3: Illustration of containers and sub-containers.

Let $h.apps$ denote the set of canonical names of the applications that are running on h , and let \mathcal{A} be the canoni-

¹More precisely, $Core(h)$ is a core constrained to contain h . That is, $Core(h) \setminus \{h\}$ may itself be minimal, but we require $h \in Core(h)$.

cal names of all of the applications. We denote with \mathcal{S} the set of sub-containers and with $m_s : \mathcal{C} \rightarrow 2^{\mathcal{S}}$ the function that maps a container to its sub-containers. The function $m_h : \mathcal{C} \times \mathcal{A} \rightarrow \mathcal{S}$ maps a container and application to a sub-container; thus, for each $a \in h.apps$, host h is in each sub-container $m_h(m_c(h.os), a)$.

At this high level of abstraction, advertising a configuration is straightforward. Initially \mathcal{C} is empty. To advertise its configuration, a host h first ensures that there is a container $c \in \mathcal{C}$ such that $m_c(h.os) = c$. Then, for each attribute $a \in h.apps$, h ensures that there is a sub-container $m_h(c, a)$ containing h .

5.2 Computing cores

The heuristics we describe in this section compute $Core(h)$ in time linear with the number of attributes in $h.apps$. These heuristics reference the set \mathcal{C} of containers and the three functions m_c , m_s and m_h , but they do not reference the full set \mathcal{A} of attributes. In addition, these heuristics do not enumerate \mathcal{H} , but they do reference the configuration of hosts (to reference the configuration of a host h' , they reference $h'.os$ and $h'.apps$). Thus, the container/sub-container hierarchy is the only data structure that the heuristics use to compute cores.

5.2.1 Metrics

We evaluate our heuristics using three metrics:

- **Average core size:** $|Core(h)|$ averaged over all $h \in \mathcal{H}$. This metric is important because it determines how much capacity is available in the system. As the average core size increases, the total capacity of the system decreases.
- **Maximum load:** The load of a host h' is the number of cores $Core(h)$ of which h' is a member. The maximum load is the largest load of any host $h' \in \mathcal{H}$.
- **Average coverage:** We say that an attribute a of a host h is *covered* in $Core(h)$ if there is at least one other host h' in $Core(h)$ that does not have a . Thus, an exploit of attribute a can affect h , but not h' , and so not all hosts in $Core(h)$ are affected. The *coverage* of $Core(h)$ is the fraction of attributes of h that are covered. The *average coverage* is the average of the coverages of $Core(h)$ over all hosts $h \in \mathcal{H}$. A high average coverage indicates a higher resilience to Internet catastrophes: many hosts have most or all of their attributes covered. We return to this discussion of what coverage means in practice in Section 5.3, after we present most of our simulation results for context.

For brevity, we use the terms core size, load, and coverage to indicate average core size, maximum load, and average coverage, respectively. Where we do refer to these terms in the context of a particular host, we say so explicitly.

	Core size	Coverage	Load
Random	5	0.977	12
Uniform	2.56	0.9997	284
Weighted	2.64	0.9995	84
DWeighted	2.58	0.9997	91

Table 3: A typical run of the heuristics.

A good heuristic will determine cores with small size, low load, and high coverage. Coverage is the most critical metric because it determines how well it does in guaranteeing service in the event of a catastrophe. Coverage may not equal 1 either because there was no host h' that was available to cover an attribute a of h , or because the heuristic failed to identify such a host h' . As shown in the following sections, the second case rarely happens with our heuristics.

Note that, as a single number, the coverage of a given $Core(h)$ does not fully capture its resilience. For example, consider host h_1 with two attributes and host h_2 with 10 attributes. If $Core(h_1)$ covers only one attribute, then $Core(h_1)$ has a coverage of 0.5. If $Core(h_2)$ has the same coverage, then it covers only 5 of the 10 attributes. There are more ways to fail all of the hosts in $Core(h_2)$ than those in $Core(h_1)$. Thus, we also use the number of cores that do not have a coverage of 1.0 as an extension of the coverage metric.

5.2.2 Heuristics

We begin by using simulation to evaluate a naive heuristic called **Random** that we use as a basis for comparison. It is not a greedy heuristic and does not reference the advertised configurations. Instead, h simply chooses at random a subset of \mathcal{H} of a given size containing h .

The first row of Table 3 shows the results of **Random** using one run of our simulator. We set the size of the cores to 5, i.e., **Random** chose 5 random hosts to form a core. The coverage of 0.977 may seem high, but there are still many cores that have uncovered attributes and choosing a core size smaller than five results in even lower coverage. The load is 12, which is significantly higher than the lower bound of 5.²

Our first greedy heuristic **Uniform** (“uniform” selection among operating systems) operates as follows. First, it chooses a host with a different operating system than $h.os$ to cover this attribute. Then, for each attribute $a \in h.apps$, it chooses both a container $c \in \mathcal{C} \setminus \{m_c(h.os)\}$ and a sub-container $sc \in m_s(c) \setminus \{m_h(c, a)\}$ at random. Finally, it chooses a host h' at random from sc . If $a \notin h'.apps$ then it includes h' in $Core(h)$. Otherwise, it tries again by choosing a new container c , sub-container sc , and host h' at random. **Uniform** repeats this procedure $diff_OS$ times in an attempt to cover a with $Core(h)$. If it fails to cover a , then the heuristic tries up to $same_OS$ times to cover a by choosing

a sub-container $sc \in m_c(h.os)$ at random and a host h' at random from sc .

The goal for having two steps, one with $diff_OS$ and another with $same_OS$, is to first exploit diversity across operating systems, and then to exploit diversity among hosts within the same operating system group. Referring back to Figure 1, the set of prevalent services among hosts running the same operating system varies across the different operating systems. In the case the attribute cannot be covered with hosts running other operating systems, the diversity within an operating system group may be sufficient to find a host h' without attribute a .

In all of our simulations, we set $diff_OS$ to 7 and $same_OS$ to 4. After experimentation, these values have provided a good trade-off between number of useless tries and obtaining good coverage. However, we have yet to study how to in general choose good values of $diff_OS$ and $same_OS$.

Pseudo-code for **Uniform** is as follows.

```

Algorithm Uniform on input  $h$ :
integer  $i$ ;
 $core \leftarrow \{h\}$ ;
 $C' \leftarrow \mathcal{C} \setminus \{m_c(h.os)\}$ 
for each attribute  $a \in h.apps$ 
   $i \leftarrow 0$ 
  while ( $a$  is not covered)  $\wedge$ 
    ( $i \leq diff\_OS + same\_OS$ )
    if ( $i \leq diff\_OS$ ) choose randomly  $c \in C'$ 
      else  $c \leftarrow m_c(h.os)$ 
    choose randomly  $sc \in m_s(c) \setminus \{m_h(c, a)\}$ 
    choose a host  $h' \in sc : h' \neq h$ 
    if ( $h'$  covers  $a$ ) add  $h'$  to  $core$ 
     $i \leftarrow i + 1$ 
return  $core$ 

```

The second row of Table 3 shows the performance of **Uniform** for a representative run of our simulator. The core size is close to the minimum size of two, and the coverage is very close to the ideal value of one. This means that using **Uniform** results in significantly better capacity and improved resilience than **Random**. On the other hand, the load is very high: there is at least one host that participates in 284 cores. The load is so high because h chooses containers and sub-containers uniformly. When constructing the cores for hosts of a given operating system, the other containers are referenced roughly the same number of times. Thus, **Uniform** considers hosts running less prevalent operating systems for inclusion in cores a disproportionately large number of times. A similar argument holds for hosts running less popular applications.

This behavior suggests refining the heuristic to choose containers and applications weighted on the popularity of their operating systems and applications. Given a container c , let $N_c(c)$ be the number of distinct hosts in the sub-containers of c , and given a set of containers C , let $N_c(C)$ be the sum of $N_c(c)$ for all $c \in C$. The heuristic **Weighted** (“weighted” OS selection) is the same as **Uniform** except that for the

²To meet this bound, number the hosts in \mathcal{H} from 0 to $|\mathcal{H}| - 1$. Let $Core(h)$ be the hosts $\{h + i \pmod{|\mathcal{H}|} : i \in \{0, 1, 2, 3, 4\}\}$.

first *diff_OS* attempts, h chooses a container c with probability $N_c(c)/N_c(\mathcal{C} \setminus \{m_c(h.os)\})$. Heuristic **DWeighted** (“doubly-weighted” selection) takes this a step further. Let $N_s(c, a)$ be $|m_h(c, a)|$ and $N_s(c, A)$ be the size of the union of $m_h(c, a)$ for all $a \in A$. Heuristic **DWeighted** is the same as **Weighted** except that, when considering attribute $a \in h.apps$, h chooses a host from sub-container $m_h(c, a')$ with probability $N_s(c, a')/N_s(c, \mathcal{A} \setminus \{a\})$.

In the third and fourth rows of Table 3, we show a representative run of our simulator for both of these variations. The two variations result in comparable core sizes and coverage as **Uniform**, but significantly reduce the load. The load is still very high, though: at least one host ends up being assigned to over 80 cores.

Another approach to avoid a high load is to simply disallow it at the risk of decreasing the coverage. That is, for some value of L , once a host h' is included in L cores, h' is removed from the structure of advertised configurations. Thus, the load of any host is constrained to be no larger than L .

What is an effective value of L that reduces load while still providing good coverage? We answer this question by first establishing a lower bound on the value of L . Suppose that a is the most prevalent attribute (either service or operating system) among all attributes, and it is present in a fraction x of the host population. As a simple application of the pigeon-hole principle, some host must be in at least l cores, where l is defined as:

$$l = \left\lceil \frac{|\mathcal{H}| \cdot x}{|\mathcal{H}| \cdot (1 - x)} \right\rceil = \left\lceil \frac{x}{(1 - x)} \right\rceil \quad (1)$$

Thus, the value of L cannot be smaller than l . Using Table 2, we have that the most prevalent attribute (port 139) is present in 55.3% of the hosts. In this case, $l = 2$.

Using simulation, we now evaluate our heuristics in terms of core size, coverage, and load as a function of the load limit L . Figures 4–7 present the results of our simulations. In these figures, we vary L from the minimum 2 through a high load of 10. All the points shown in these graphs are the averages of eight simulated runs with error bars (although they are too narrow to be seen in some cases). For Figures 4–6, we use the standard error to determine the limits of the error bars, whereas for Figure 7 we use the maximum and minimum observed among our samples. When using load limit as a threshold, the order in which hosts request cores from \mathcal{H} will produce different results. In our experiments, we randomly choose eight different orders of enumerating \mathcal{H} for constructing cores. For each heuristic, each run of the simulator uses a different order. Finally, we vary the core size of **Random** using the load limit L to illustrate its effectiveness across a range of core sizes.

Figure 4 shows the average core size for the four algorithms for different values of L . According to this graph, **Uniform**, **Weighted**, and **DWeighted** do not differ much in terms of

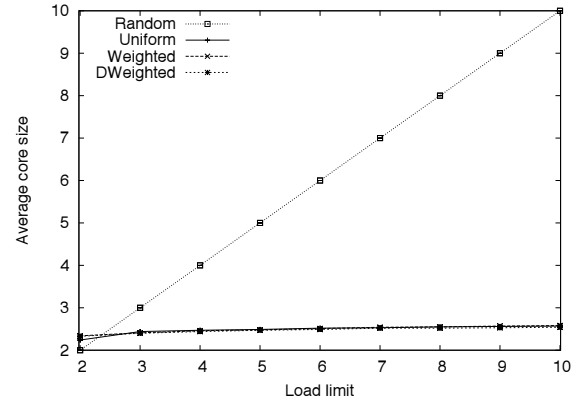


Figure 4: Average core size.

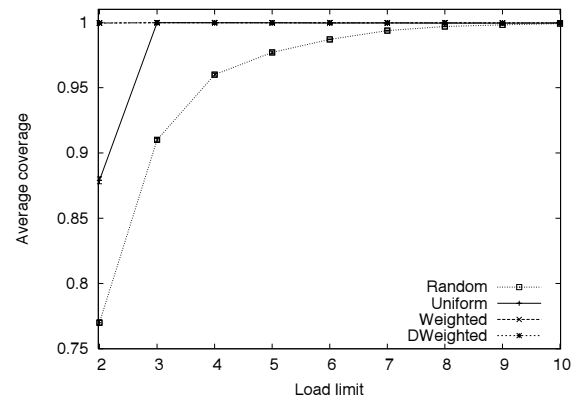


Figure 5: Average coverage.

core size. The average core size of **Random** increases linearly with L by design.

In Figure 5, we show results for coverage. Coverage is slightly smaller than 1.0 for **Uniform**, **Weighted**, and **DWeighted** when L is greater or equal to three. For $L = 2$, **Weighted** and **DWeighted** still have coverage slightly smaller than 1.0, but **Uniform** does significantly worse. Using weighted selection is useful when L is small. **Random** improves coverage with increasing L because the size of the cores increases. Note that, to reach the same value of coverage obtained by the other heuristics, **Random** requires a large core size of 9.

There are two other important observations to make about this graph. First, coverage is roughly the same for **Uniform**, **Weighted**, and **DWeighted** when $L > 2$. Second, as L continues to increase, there is a small decrease in coverage. This is due to the nature of our traces and to the random choices made by our algorithms. Ports such as 111 (portmapper, rpcbind) and 22 (sshd) are open on several of the hosts with operating systems different than Windows. For small values of L , these hosts rapidly reach their threshold. Consequently, when hosts that do have these services as attributes request a core, there are fewer hosts available with these same at-

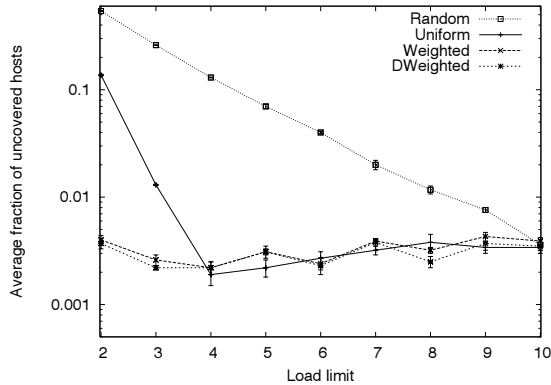


Figure 6: Average fraction of uncovered hosts.

tributes. On the other hand, for larger values of L , these hosts are more available, thus slightly increasing the probability that not all the attributes are covered for hosts executing an operating system different than Windows. We observed this phenomenon exactly with ports 22 and 111 in our traces.

This same phenomenon can be observed in Figure 6. In this figure, we plot the average fraction of hosts that are not fully covered, which is an alternative way of visualizing coverage. We observe that there is a share of the population of hosts that are not fully covered, but this share is very small for **Uniform** and its variations. Such a set is likely to exist due to the non-deterministic choices we make in our heuristics when forming cores. These uncovered hosts, however, are not fully unprotected. From our simulation traces, we note the average number of uncovered attributes is very small for **Uniform** and its variations. In all runs, we have just a few hosts that do not have all their attributes covered, and in the majority of the instances there is just a single uncovered attribute.

Finally, we show the resulting variance in load. Since the heuristics limit each host to be in no more than L cores, the maximum load equals L . The variance indicates how fairly the load is spread among the hosts. As expected, **Random** does well, having the lowest variance among all the algorithms and for all values of L . Ordering the greedy heuristics by their variance in load, we have **Uniform** \succ **Weighted** \succ **DWeighted**. This is not surprising since we introduced the weighted selection exactly to better balance the load. It is interesting to observe that for every value of L , the load variance obtained for **Uniform** is close to L . This means that there were several hosts not participating in any core and several other hosts participating in L cores.

A larger variance in load may not be objectionable in practice as long as a maximum load is enforced. Given the extra work of maintaining the functions N_s and N_c , the heuristic **Uniform** with small L ($L > 2$) is the best choice for our application. However, should load variance be an issue, we can use one of the other heuristics.

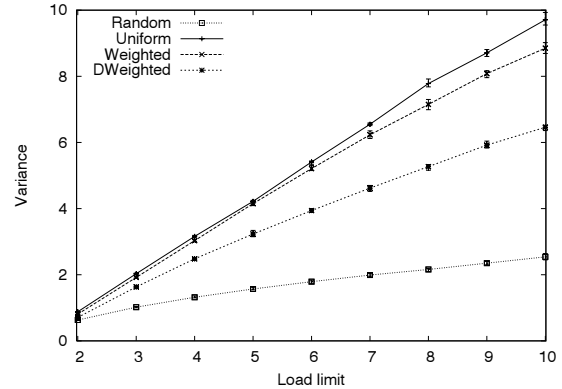


Figure 7: Average load variance.

5.3 Translating to real pathogens

In this section, we discuss why we have chosen to tolerate exploits of vulnerabilities on a single attribute at a time. We do so based on information about past worms to support our choices and assumptions.

Worms such as the ones in Table 1 used services that have vulnerabilities as vectors for propagation. Code Red, for example, used a vulnerability in the IIS Web server to infect hosts. In this example, a vulnerability on a single attribute (Web server listening on port 80) was exploited. In other instances, such as with the Nimda worm, more than one vulnerability was exploited during propagation, such as via e-mail messages and Web browsing. Although these cases could be modeled as exploits to vulnerabilities on multiple attributes, we observe that previous worms did not propagate across operating system platforms: in fact, the worms targeted services on various versions of Windows.

By covering classes of operating systems in our cores, we guarantee that pathogens that exploit vulnerabilities on a single platform are not able to compromise all the members of a core C of a particular host h , assuming that C covers all attributes of h . Even if $Core(h)$ leaves some attributes uncovered, h is still protected against attacks targeting covered attributes. Referring back to Figure 6, the majority of the cores have maximum coverage. We also observed in the previous section that, for cores that do not have maximum coverage, usually it is only a single uncovered attribute.

Under our assumptions, informed replication mitigates the effects of a worm that exploits vulnerabilities on a service that exists across multiple operating systems, and of a worm that exploits vulnerabilities on services in a single operating system. Figure 6 presents a conservative estimate on the percentage of the population that is unprotected in the case of an outbreak of such a pathogen. Assuming conservatively that every host that is not fully covered has the same uncovered attribute, the numbers in the graph give the fraction of the population that can be affected in the case of an outbreak. As can be seen, this fraction is very small.

With our current use of attributes to represent software heterogeneity, a worm can be effective only if it can exploit vulnerabilities in services that run across operating systems, or if it exploits vulnerabilities in multiple operating systems. To the best of our knowledge, there has been no large-scale outbreak of such a worm. Of course, such a worm could be written. In the next section, we discuss how to modify our heuristics to cope with exploits of vulnerabilities on multiple attributes.

5.4 Exploits of multiple attributes

To tolerate exploits on multiple attributes, we need to construct cores such that, for subsets of attributes possessed by members of a core, there must be a core member that does not have these attributes. We call a k -resilient core C a group of hosts in \mathcal{H} such that, for every k attributes of members of C , there is at least one host in C that does not contain any of these attributes. In this terminology, the cores we have been considering up to this point have been 1-resilient cores.

To illustrate this idea, consider the following example. Hosts run *Windows*, *Linux*, and *Solaris* as operating systems, and *IIS*, *Apache*, and *Zeus* as Web servers. An example of a 2-resilient core is a subset composed of hosts h_1, h_2, h_3 with configurations: $h_1 = \{\text{Linux}, \text{Apache}\}$; $h_2 = \{\text{Windows}, \text{IIS}\}$; $h_3 = \{\text{Solaris}, \text{Zeus}\}$. In this core, for every pair of attributes, there is at least one host that contains none of them.

As before, every host h builds a k -resilient core $\text{Core}(h)$. To build $\text{Core}(h)$, host h uses the following heuristic:

Step 1 Select randomly $k - 1$ hosts, h_1 through h_{k-1} , such that $h_{i.os} = h.os$, for every $i \in \{1, \dots, k - 1\}$;

Step 2 Use **Uniform** to search for a 1-resilient core C for h ;

Step 3 For each $i \in \{1, \dots, k - 1\}$, use **Uniform** to search for a 1-resilient core C_i for h_i ;

Step 4 $\text{Core}(h) \leftarrow C \cup C_1 \cup \dots \cup C_{k-1}$.

Intuitively, to form a k -resilient core we need to gather enough hosts such that we can split these hosts into k subsets, where at least one subset is a 1-resilient core. Moreover, if there are two of these subsets where, for each subset, all of the members of that subset share some attribute, then the shared attribute of one set must be different from the shared attribute of the other set. Our heuristic is conservative in searching independently for 1-resilient cores because the problem does not require all such sets to be 1-resilient cores. In doing so, we protect clients and at the same time avoid the complexity of optimally determining such sets. The sets output by the heuristic, however, may not be minimal, and therefore they are approximations of theoretical cores. We discuss this heuristic further in [13].

In Table 4, we show simulation results for this heuristic for $k = 2$. The first column shows the values of load limit (L) used by the **Uniform** heuristic to compute cores. We chose

L	Avg. 2-coverage	Avg. 1-coverage	Avg. Core size
5	0.829 (0.002)	0.855 (0.002)	4.19 (0.004)
6	0.902 (0.002)	0.917 (0.002)	4.59 (0.005)
7	0.981 (0.001)	0.987 (0.001)	5.00 (0.005)
8	0.995 (0.0)	1.0 (0.0)	5.11 (0.005)
9	0.996 (0.0)	1.0 (0.0)	5.14 (0.005)
10	0.997 (0.0)	1.0 (0.0)	5.17 (0.003)

Table 4: Summary of simulation results for $k = 2$ for 8 different runs.

values of $L \geq 5$ based on an argument generalized from the one given in Section 5.2 giving the lower bound of L [13]. In the second and third columns, we present our measurements for coverage with standard error in parentheses. For each computed core $\text{Core}(h)$, we calculate the fraction of pairs of attributes such that at least one host $h' \in \text{Core}(h)$ contains none of attributes of the pair. We name this metric **2-coverage**, and in the table we present the average across all hosts and across all eight runs of the simulator. **1-coverage** is the same as the average coverage metric defined in Section 5.2. Finally, the last column shows average core size.

According to the coverage results, the heuristic does well in finding cores that protect hosts against potential pathogens that exploit vulnerabilities in at most two attributes. A beneficial side-effect of protecting against exploits on two attributes is that the amount of diversity in a 2-resilient core permits better protection to its client against pathogens that exploit vulnerabilities on single attributes. For values of L greater than seven, all clients have all their attributes covered (the average 1-coverage metric is one and the standard error is zero).

Having a system that more broadly protects its hosts requires more resources: core sizes are larger to obtain sufficiently high degrees of coverage. Compared to the results in Section 5.2, we observe that we need to double the load limit to obtain similar values for coverage. This is not surprising. In our heuristic, for each host, we search for two 1-resilient cores. We therefore need to roughly double the amount of resources used.

Of course, there is a limit to what can be done with informed replication. As k increases, the demand on resources continues to grow, and a point will be reached in which there is not enough diversity to withstand an attack that targets $k+1$ attributes. Using our diversity study results in Table 2, if a worm were able to simultaneously infect machines that run one of the first four operating systems in this table, the worm could potentially infect 84% of the population. The release of such a worm would most likely cause the Internet to collapse. An approach beyond informed replication would be needed to combat an act of cyberterrorism of this magnitude.

6 The Phoenix Recovery Service

A cooperative recovery service is an attractive architecture for tolerating Internet catastrophes. It is attractive for both individual Internet users, like home broadband users, who do not

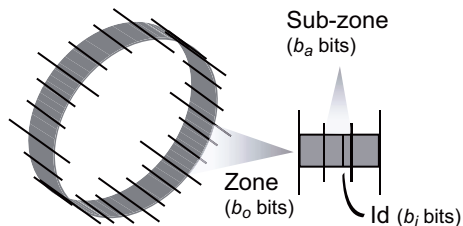


Figure 8: Phoenix ring.

wish to pay for commercial backup service or deal with the inconvenience of making manual backups, as well as corporate environments, which often have a significant amount of unused disk space per machine. If Phoenix were deployed, users would not need to exert significant effort to backup their data, and they would not require local backup systems. Phoenix makes specifying what data to protect as straightforward as specifying what data to share on file-sharing peer-to-peer systems. Further, a cooperative architecture has little cost in terms of time and money; instead, users relinquish a small fraction of their disk, CPU, and network resources to gain access to a highly resilient backup service.

As with Pastiche [8], we envision using Phoenix as a cooperative recovery service for user data. However, rather than exploiting redundant data on similar hosts to reduce backup costs for operating system and application software, we envision Phoenix users only backing up user-generated data and relying upon installation media to recover the operating system and application software. With this usage model, broadband users of Phoenix can recover 10 GB of user-generated data in a day. Given the relatively low capacity utilization of disks in desktop machines [3], 10 GB should be sufficient for a wide range of users. Further, users can choose to be more selective in the data backed up to reduce their recovery time. We return to the issue of bandwidth consumption and recovery time in Section 7.3.

6.1 System overview

A Phoenix host selects a subset of hosts to store backup data, expecting that at least one host in the subset survives an Internet catastrophe. This subset is a core, chosen using the **Uniform** heuristic described above.

Choosing cores requires knowledge of host software configurations. As described in Section 5, we use the container mechanism for advertising configurations. In our prototype, we implement containers using the Pastry [32] distributed hash table (DHT). Pastry is an overlay of nodes that have identifiers arranged in a ring. This overlay provides a scalable mechanism for routing requests to appropriate nodes.

Phoenix structures the DHT identifier space hierarchically. It splits the identifier space into *zones*, mapping containers to zones. It further splits zones into *sub-zones*, mapping sub-containers to equally-sized sub-zones. Figure 8 illustrates this

hierarchy. Corresponding to the hierarchy, Phoenix creates host identifiers out of three parts. To generate its identifier, a host concatenates the hash representing its operating system $h.os$, the hash representing an attribute $a \in h.apps$, and the hash representing its IP address. As Figure 8 illustrates, each part has b_o , b_a , and b_i bits, respectively. To advertise its configuration, a host creates a hash for each one of its attributes. It therefore generates as many identifiers as the number of attributes in $h.apps$. It then joins the DHT at multiple points, each point being characterized by one of these identifiers. Since the hash of the operating system is the initial, “most significant” part of all the host’s identifiers, all identifiers of a host lie within the same zone.

To build $Core(h)$ using **Uniform**, host h selects hosts at random. When trying to cover an attribute a , h first selects a container at random, which corresponds to choosing a number c randomly from $[0, 2^{b_o} - 1]$. The next step is to select a sub-container and a host within this sub-container both at random. This corresponds to choosing a random number sc within $[0, 2^{b_a} - 1]$ and another random number id within $[0, 2^{b_i} - 1]$, respectively. Host h creates a Phoenix identifier by concatenating these various components as $(c \circ sc \circ id)$. It then performs a lookup on the Pastry DHT for this identifier. The host h' that satisfies this lookup informs h of its own configuration. If this configuration covers attribute a , h adds h' to its core. If not, h repeats this process.

The hosts in h ’s core maintain backups of its data. These hosts periodically send announcements to h . In the event of a catastrophe, if h loses its data, it waits for one of these announcements from a host in its core, say h' . After receiving such a message, h requests its data from h' . Since recovery is not time-critical, the period between consecutive announcements that a host sends can be large, from hours to a day.

A host may permanently leave the system after having backed up its files. In this situation, other hosts need not hold any backups for this host and can use garbage collection to retrieve storage used for the departed host’s files. Thus, Phoenix hosts assume that if they do not receive an acknowledgment for any announcement sent for a large period of time (e.g., a week), then this host has left the system and its files can be discarded.

Since many hosts share the same operating systems, Phoenix identifiers are not mapped in a completely random fashion into the DHT identifier space. This could lead to some hosts receiving a disproportionate number of requests. For example, consider a host h that is either the first of a populated zone that follows an empty zone or is the last host of a populated zone that precedes an empty zone. Host h receives requests sent to the empty zone because, by the construction of the ring, its address space includes addresses of the empty zone. In our design, however, once a host reaches its load limit, it can simply discard new requests by the Phoenix protocol.

Experimenting with the Phoenix prototype, we found that

constructing cores performed well even with an unbalanced ID space. But a simple optimization can improve core construction further. The system can maintain an OS hint list that contains canonical names of operating systems represented in the system. When constructing a core, a host then uses hashes of these names instead of generating a random number. Such a list could be maintained externally or generated by sampling. We present results for both approaches in Section 7.

We implemented Phoenix using the Macedon [31] framework for implementing overlay systems. The Phoenix client on a host takes a tar file of data to be backed up as input together with a host configuration. In the current implementation, users manually specify the host configuration. We are investigating techniques for automating the configuration determination, but we expect that, from a practical point of view, a user will want to have some say in which attributes are important.

6.2 Attacks on Phoenix

Phoenix uses informed replication to survive wide-spread failures due to exploits of vulnerabilities in unrelated software on hosts. However, Phoenix itself can also be the target of attacks mounted against the system, as well as attacks from within by misbehaving peers.

The most effective way to attack the Phoenix system as a whole is to unleash a pathogen that exploits a vulnerability in the Phoenix software. In other words, Phoenix itself represents a shared vulnerability for all hosts running the service. This shared vulnerability is not a covered attribute, hence an attack that exploits a vulnerability in the Phoenix software would make it possible for data to be lost as a pathogen spreads unchecked through the Phoenix system. To the extent possible, Phoenix relies on good programming practices and techniques to prevent common attacks such as buffer overflows. However, this kind of attack is not unique to Phoenix or the use of informed replication. Such an attack is a general problem for any distributed system designed to protect data, even those that use approaches other than informed replication [11]. A single system fundamentally represents a shared vulnerability; if an attacker can exploit a vulnerability in system software and compromise the system, the system cannot easily protect itself.

Alternatively, hosts participating in Phoenix can attack the system by trying to access private data, tamper with data, or mount denial-of-service attacks. To prevent malicious servers from accessing data without authorization or from tampering with data, we can use standard cryptographic techniques [13]. In particular, we can guarantee the following: (1) the privacy and integrity of any data saved by any host is preserved, and (2) if a client host contacts an honest server host for a backup operation, then the client is able to recover its data after a catastrophe. From a security perspective, the most relevant part of the system is the interaction process between a host

client and a host server which has agreed to participate in the host's core.

Malicious servers can mount a denial-of-service attack against a client by agreeing to hold a replica copy of the client's data, and subsequently dropping the data or refusing recovery requests. One technique to identify such misbehaviors is to issue *signed receipts* [13]. Clients can use such receipts to claim that servers are misbehaving. As we mentioned before, servers cannot corrupt data assuming robustness of the security primitives.

Hosts could also advertise false configurations in an attempt to free-ride in the system. By advertising attributes that make a host appear more unreliable, the system will consider the host for fewer cores than otherwise. As a result, a host may be able to have its data backed up without having to back up its share of data.

To provide a disincentive against free-riders, members of a core can maintain the configuration of hosts they serve, and serve a particular client only if their own configuration covers at least one client attribute. By sampling servers randomly, it is possible to reconstruct cores and eventually find misbehaving clients.

An important feature of our heuristic that constrains the impact of malicious hosts on the system is the load limit: if only a small percentage of hosts is malicious at any given time, then only a small fraction of hosts are impacted by the maliciousness. Hosts not respecting the limit can also be detected by random sampling.

7 Phoenix evaluation

In this Section, we evaluate our Phoenix prototype on the PlanetLab testbed using the metrics discussed in Section 5. We also simulate a catastrophic event — the simultaneous failure of all Windows hosts — to experiment with Phoenix's ability to recover from large failures. Finally, we discuss the time and bandwidth required to recover from catastrophes.

7.1 Prototype evaluation

We tested our prototype on 63 hosts across the Internet: 62 PlanetLab hosts and one UCSD host. To simulate the diversity we obtained in the study presented in Section 4, we selected 63 configurations at random from our set of 2,963 configurations of general-purpose hosts, and made each of these configurations an input to the Phoenix service on a host. In the population we have chosen randomly, out of the 63 configurations 38 have Windows as their operating system. Thus, in our setting roughly 60% of the hosts represent Windows hosts. From Section 5.2, the load limit has to be at least three.

For the results we present in this section, we use an OS hint list while searching for cores. Varying L , we obtained the values in Table 5 for coverage, core size, and load variance for a representative run of our prototype. For comparison, we

Load limit (L)	Core size		Coverage		Load var.	
	Imp.	Sim.	Imp.	Sim.	Imp.	Sim.
3	2.12	2.23	1.0	1.0	1.65	1.88
5	2.10	2.25	1.0	1.0	2.88	3.31
7	2.10	2.12	1.0	1.0	4.44	3.56

Table 5: Implementation results on PlanetLab (“Imp”) with simulation results for comparison (“Sim”).

also present results from our simulations with the same set of configurations used for the PlanetLab experiment. From the results in the table, coverage is perfect in all cases, and the average core size is less than 3 (less than 2 replica copies).

The major difference in increasing the value of L is the respective increase in load variance. As L increases, load balance worsens. We also counted the number of requests issued by each host in its search for a core. Different from our simulations, we set a large upper bound on the number of request messages ($diff_OS + same_OS = 100$) to verify the average number of requests necessary to build a core, and we had hosts searching for other hosts only outside their own zones ($same_OS = 0$). The averages for number of requests are 14.6, 5.2, and 4.1 for values of L of 3, 5, and 7, respectively. Hence, we can tradeoff load balance and message complexity.

We also ran experiments without using an OS hint list. The results are very good, although worse than the implementation that uses hint lists. We observed two main consequences in not using a hint list. First, the average number of requests is considerably higher (over 2x). Second, for small values of L ($L = 3, 5$), some hosts did not obtain perfect coverage.

7.2 Simulating catastrophes

Next we examine how the Phoenix prototype behaves in a severe catastrophe: the exploitation and failure of all Windows hosts in the system. This scenario corresponds to a situation in which a worm exploits a vulnerability present in *all* versions of Windows, and corrupts the data on the compromised hosts. Note that this scenario is far more catastrophic than what we have experienced with worms to date. The worms listed in Table 1, for example, exploit only particular services on Windows.

The simulation proceeded as follows. Using the same experimental setting as above, hosts backed up their data under a load limit constraint of $L = 3$. We then triggered a failure in all Windows hosts, causing the loss of data stored on them. Next we restarted the Phoenix service on the hosts, causing them to wait for announcements from other hosts in their cores (Section 6.1). We then observed which Windows hosts received announcements and successfully recovered their data.

All 38 hosts recovered their data in a reasonable amount of time. For 35 of these hosts, it took on average 100 seconds to recover their data. For the other three machines, it took several minutes due to intermittent network connectivity

(these machines were in fact at the same site). Two important parameters that determine the time for a host to recover are the frequency of announcements and the backup file size (transfer time). We used an interval between two consecutive announcements to the same client of 120 seconds, and a total data size of 5 MB per host. The announcement frequency depends on the user expectation on recovery speed. In our case, we wanted to finish each experiment in a reasonable amount of time. Yet, we did not want to have hosts sending a large number of announcement messages unnecessarily. For the backup file size, we chose an arbitrary value since we are not concerned about transfer time in this experiment. On the other hand, this size was large enough to hinder recovery when connectivity between client and server was intermittent.

It is important to observe that we stressed our prototype by causing the failure of these hosts almost simultaneously. Although the number of nodes we used is small compared to the potential number of nodes that Phoenix can have as participants, we did not observe any obvious scalability problems. On the contrary, the use of a load limit helped in constraining the amount of work a host does for the system, independent of system size.

7.3 Recovering from a catastrophe

We now examine the bandwidth requirements for recovering from an Internet catastrophe. In a catastrophe, many hosts will lose their data. When the failed hosts come online again, they will want to recover their data from the remaining hosts that survived the catastrophe. With a large fraction of the hosts recovering simultaneously, a key question is what bandwidth demands the recovering hosts will place on the system.

The aggregate bandwidth required to recover from a catastrophe is a function of the amount of data stored by the failed hosts, the time window for recovery, and the fraction of hosts that fail. Consider a system of 10,000 hosts that have software configurations analogous to those presented in Section 4, where 54.1% of the hosts run Windows and the remaining run some other operating system. Next consider a catastrophe similar to the one above in which all Windows hosts, independent of version, lose the data they store. Table 6 shows the bandwidth required to recover the Windows hosts for various storage capacities and recovery periods. The first column shows the average amount of data a host stores in the system. The remaining columns show the bandwidth required to recover that data for different periods.

The first four rows show the aggregate system bandwidth required to recover the failed hosts: the total amount of data to recover divided by the recovery time. This bandwidth reflects the load on the network during recovery. Assuming a deployment over the Internet, even for relatively large backup sizes and short recovery periods, this load is small. Note that these results are for a system with 10,000 hosts and that, for an equivalent catastrophe, the aggregate bandwidth require-

Size (GB)	1 hour	1 day	1 week
Aggregate bandwidth			
0.1	1.2 Gb/s	50 Mb/s	7.1 Mb/s
1	12 Gb/s	0.50 Gb/s	71 Mb/s
10	120 Gb/s	5.0 Gb/s	710 Mb/s
100	1.2 Tb/s	50 Gb/s	7.1 Gb/s
Per-host bandwidth ($L = 3$)			
0.1	0.7 Mb/s	28 Kb/s	4.0 Kb/s
1	6.7 Mb/s	280 Kb/s	40 Kb/s
10	66.7 Mb/s	2.8 Mb/s	400 Kb/s
100	667 Mb/s	28 Mb/s	4.0 Mb/s

Table 6: Bandwidth consumption after a catastrophe.

ments will scale linearly with the number of hosts in the system and the amount of data backed up.

The second four rows show the average per-host bandwidth required by the hosts in the system responding to recovery requests. Recall that the system imposes a load limit L that caps the number of replicas any host will store. As a result, a host recovers at most L other hosts. Note that, because of the load limit, per-host bandwidth requirements for hosts involved in recovery are independent of both the number of hosts in the system and the number of hosts that fail.

The results in the table show the per-host bandwidth requirements with a load limit $L = 3$, where each host responds to at most three recovery requests. The results indicate that Phoenix can recover from a severe catastrophe in reasonable time periods for useful backup sizes. As with other cooperative backup systems like Pastiche [8], per-host recovery time will depend significantly on the connectivity of hosts in the system. For example, hosts connected by modems can serve as recovery hosts for a modest amount of backed up data (28 Kb/s for 100 MB of data recovered in a day). Such backup amounts would only be useful for recovering particularly critical data, or recovering frequent incremental backups stored in Phoenix relative to infrequent full backups using other methods (e.g., for users who take monthly full backups on media but use Phoenix for storing and recovering daily incrementals). Broadband hosts can recover failed hosts storing orders of magnitude more data (1–10 GB) in a day, and high-bandwidth hosts can recover either an order magnitude more quickly (hours) or even an order of magnitude more data (100 GB). Further, Phoenix could potentially exploit the parallelism of recovering from all surviving hosts in a core to further reduce recovery time.

Although there is no design constraint on the amount of data hosts back up in Phoenix, for current disk usage patterns, disk capacities, and host bandwidth connectivity, we envision users typically storing 1–10 GB in Phoenix and waiting a day to recover their data. According to a recent study, desktops with substantial disks (> 40 GB) use less than 10% of their local disk capacity, and operating system and temporary user files consume up to 4 GB [3]. Recovery times on the order of a day are also practical. For example, previous worm catas-

trophes took longer than a day for organizations to recover, and recovery using organization backup services can take a day for an administrator to respond to a request.

8 Conclusions

In this paper, we proposed a new approach called informed replication for designing distributed systems to survive Internet epidemics that cause catastrophic damage. Informed replication uses a model of correlated failures to exploit software diversity, providing high reliability with low replication overhead. Using host diversity characteristics derived from a measurement study of hosts on the UCSD campus, we developed and evaluated heuristics for determining the number and placement of replicas that have a number of attractive features. Our heuristics provide excellent reliability guarantees (over 0.99 probability that user data survives attacks of single- and double-exploit pathogens), result in low degree of replication (less than 3 copies for single-exploit pathogens; less than 5 copies for double-exploit pathogens), limit the storage burden on each host in the system, and lend themselves to a fully distributed implementation. We then used this approach in the design and implementation of a cooperative backup system called the Phoenix Recovery Service. Based upon our evaluation results, we conclude that our approach is a viable and attractive method for surviving Internet catastrophes.

Acknowledgements

We would like to express our gratitude to Pat Wilson and Joe Pomianek for providing us with the UCSD host traces. We would also like to thank Chip Killian for his valuable assistance with Macedon, Marvin McNett for system support for performing our experiments, Stefan Savage for helpful discussions, and our shepherd Steve Gribble, whose comments significantly improved this paper. Support for this work was provided in part by AFOSR MURI Contract F49620-02-1-0233 and DARPA FTN Contract N66001-01-1-8933.

References

- [1] E. G. Barrantes et al. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM CCS*, pages 281–289, Washington D.C., USA, Oct. 2003.
- [2] C. Batten, K. Barr, A. Saraf, and S. Treptin. pStore: A secure peer-to-peer backup system. Technical Report MIT-LCS-TM-632, MIT, Dec. 2001.
- [3] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu. Kosha: A peer-to-peer enhancement for the network file system. In *Proc. of ACM/IEEE Supercomputing*, Pittsburgh, PA, Nov 2004.
- [4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of the 3rd ACM/USENIX OSDI*, pages 173–186, New Orleans, LA, Feb. 1999.

- [5] Y. Chen. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, pages 28–37, Berkeley, CA, Aug. 1999.
- [6] Codegreen. <http://www.winnetmag.com/Article/ArticleID/22381/22381.html>.
- [7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan. 1998.
- [8] L. P. Cox and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. of the 5th ACM/USENIX OSDI*, pages 285–298, Boston, MA, Dec. 2002.
- [9] CRclean. <http://www.winnetmag.com/Article/ArticleID/22381/22381.html>.
- [10] H. A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proc. of the 13th USENIX Security Symposium*, pages 271–286, Aug. 2004.
- [11] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of NSDI*, May 2005.
- [12] Insecure.org. The nmap tool. <http://www.insecure.org/nmap>.
- [13] F. Junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. M. Voelker. Coping with internet catastrophes. Technical Report CS2005–0816, UCSD, Feb 2005.
- [14] F. Junqueira, R. Bhagwan, K. Marzullo, S. Savage, and G. M. Voelker. The Phoenix Recovery System: Rebuilding from the ashes of an Internet catastrophe. In *Proc. of HotOS-IX*, pages 73–78, Lihue, HI, May 2003.
- [15] F. Junqueira and K. Marzullo. Synchronous Consensus for dependent process failures. In *Proc. of ICDCS*, pages 274–283, Providence, RI, May 2003.
- [16] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *Proceedings of the 4th Virus Bulletin International Conference*, pages 178–184, Abingdon, England, 1994.
- [17] C. Kreibich and J. Crowcroft. Honeycomb – Creating intrusion detection signatures using honeypots. In *Proc. of HotNets-II*, pages 51–56, Cambridge, MA, Nov. 2003.
- [18] J. Kubiawicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ACM ASPLOS*, pages 190–201, Cambridge, MA, 2000.
- [19] J. Levin, R. LaBella, H. Owen, D. Contis, and B. Culver. The use of honeynets to detect exploited systems across large enterprise networks. In *Proc. of the IEEE Information Assurance Workshop*, pages 92–99, Atlanta, GA, June 2003.
- [20] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative Internet backup scheme. In *Proc. of USENIX Annual Technical Conference*, pages 29–42, San Antonio, TX, 2003.
- [21] T. Liston. Welcome To My Tarpit: The Tactical and Strategic Use of LaBrea. Technical report, 2001. <http://www.threenorth.com/LaBrea/LaBrea.txt>.
- [22] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks. Internet worm and virus protection in dynamically reconfigurable hardware. In *Proc. of MAPLD*, Sept. 2003.
- [23] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. of ACM STOC*, pages 569–578, El Paso, TX, 1997.
- [24] Microsoft Corporation. Microsoft windows update. <http://windowsupdate.microsoft.com>.
- [25] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Privacy & Security*, 1(4):33–39, Jul 2003.
- [26] D. Moore and C. Shannon. The spread of the Witty worm. <http://www.caida.org/analysis/security/sitty/>.
- [27] D. Moore, C. Shannon, and J. Brown. Code Red: A case study on the spread and victims of an Internet worm. In *Proc. of ACM IMW*, pages 273–284, Marseille, France, Nov. 2002.
- [28] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proc. of IEEE Infocom*, pages 1901–1910, San Francisco, CA, Apr. 2003.
- [29] D. Moore, G. M. Voelker, and S. Savage. Inferring Internet denial of service activity. In *Proc. of the USENIX Security Symposium*, Washington, D.C., Aug. 2001.
- [30] Lurhq. mydoom word advisory. <http://www.lurhq.com/mydoomadvisory.html>.
- [31] A. Rodriguez, C. Killian, S. Bhat, D. Kestic, and A. Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proc. of NSDI*, pages 267–280, San Francisco, CA, Mar 2004.
- [32] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. of ACM Middleware*, pages 329–350, Heidelberg, Germany, Nov. 2001.
- [33] S. Sidiropoulos and A. D. Keromytis. A network worm vaccine architecture. In *Proc. of IEEE Workshop on Enterprise Security*, pages 220–225, Linz, Austria, June 2003.
- [34] S. Singh, C. Estant, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Proc. of the 6th ACM/USENIX OSDI*, pages 45–60, San Francisco, CA, Dec. 2004.
- [35] Lurhq. sobig.a and the spam you received today. <http://www.lurhq.com/sobig.html>.
- [36] Sophos anti-virus. W32/sasser-a worm analysis. <http://www.sophos.com/virusinfo/analyses/w32sasser.html>, May 2004.
- [37] S. Staniford. Containment of Scanning Worms in Enterprise Networks. *Journal of Computer Security*, 2004.
- [38] Symantec. Symantec Security Response. <http://securityresponse.symantec.com/>.
- [39] T. Toth and C. Kruegel. Connection-history Based Anomaly Detection. Technical Report TUV-1841-2002-34, Technical University of Vienna, June 2002.
- [40] J. Twycross and M. M. Williamson. Implementing and testing a virus throttle. In *Proc. of the 12th USENIX Security Symposium*, pages 285–294, Washington, D.C., Aug. 2003.
- [41] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First step towards automated detection of buffer overrun vulnerabilities. In *Proc. of NDSS*, pages 3–17, San Diego, CA, Feb. 2000.
- [42] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proc. of ACM SIGCOMM*, pages 193–204, Portland, Oregon, Aug. 2004.
- [43] H. Weatherspoon, T. Mosciovitz, and J. Kubiawicz. Intropective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proc. of International Workshop on Reliable Peer-to-Peer Distributed Systems*, Oct. 2002.
- [44] M. Williamson. Throttling Viruses: Restricting propagation to defeat malicious mobile code. Technical Report HPL-2002-172, HP Laboratories Bristol, June 2002.
- [45] C. Wong et al. Dynamic quarantine of Internet worms. In *Proc. of DSN*, pages 73–82, Florence, Italy, June 2004.
- [46] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and early warning for Internet worms. In *Proceedings of the 10th ACM CCS*, pages 190–199, Washington D.C., USA, Oct. 2003.

Making Scheduling “Cool”: Temperature-Aware Workload Placement in Data Centers*

Justin Moore[†]

Jeff Chase[†]

Parthasarathy Ranganathan[‡]

Ratnesh Sharma[‡]

[†]*Department of Computer Science
Duke University
{justin,chase}@duke.edu*

[‡]*Internet Systems and Storage Lab
Hewlett Packard Labs
{partha.ranganathan, ratnesh.sharma}@hp.com*

Abstract

Trends towards consolidation and higher-density computing configurations make the problem of heat management one of the critical challenges in emerging data centers. Conventional approaches to addressing this problem have focused at the facilities level to develop new cooling technologies or optimize the delivery of cooling. In contrast to these approaches, our paper explores an alternate dimension to address this problem, namely a systems-level solution to control the heat generation through *temperature-aware workload placement*.

We first examine a theoretic thermodynamic formulation that uses information about steady state hot spots and cold spots in the data center and develop real-world scheduling algorithms. Based on the insights from these results, we develop an alternate approach. Our new approach leverages the non-intuitive observation that the source of cooling inefficiencies can often be in locations spatially uncorrelated with its manifested consequences; this enables additional energy savings. Overall, our results demonstrate up to a factor of two reduction in annual data center cooling costs over location-agnostic workload distribution, purely through software optimizations without the need for any costly capital investment.

1 Introduction

The last few years have seen a dramatic increase in the number, size, and uses of data centers. Large data centers contain up to tens of thousands of servers and support hundreds or thousands of users. For such data centers, in addition to traditional IT infrastructure issues, designers increasingly need to deal with issues of power consumption, heat dissipation, and cooling provisioning.

These issues, though traditionally the domain of facilities management, have become important to address at the IT level because of their implications on cost, reliability, and dynamic response to data center events. For example, the total cooling costs for large data centers (30,000 ft^2) can run into the tens of millions of dollars. Similarly, brownouts or cooling failures can lead to a reduced mean time between failure and service outages, as servers that overheat will automatically shut down. Furthermore, increases in server utilization [7, 16] or the failure of a CRAC unit can upset the current environment in a matter of minutes or even seconds, requiring rapid response strategies, often faster than what is possible at a facilities level. These conditions will accelerate as processor densities increase, administrators replace 1U servers with blades, and organizations consolidate multiple clusters into larger data centers.

Current work in the field of thermal management explores efficient methods of extracting heat from the data center [23, 27]. In contrast, our work explores *temperature-aware workload placement* algorithms. This approach focuses on scheduling workloads in a data center — and the resulting heat the servers generate — in a manner that minimizes the energy expended by the cooling infrastructure, leading to lower cooling costs and increased hardware reliability.

We develop temperature-aware workload placement algorithms and present the first comprehensive exploration of the benefits from these policies. Using simple methods of observing hot air flow within a data center, we formulate two workload placement policies: *zone-based discretization* (ZBD) and *minimize-heat-recirculation* (MINHR). These algorithms establish a prioritized list of servers within the data center, simplifying the task of applying these algorithms to real-work systems.

The first policy leverages a theoretic thermodynamic formulation based on steady-state hot spots and cold spots in

*This work is supported in part by HP Labs, and the U.S. National Science Foundation (EIA-9972879, ANI-0330658, and ANI-0126231).

the data center [27]. The second policy uses a new formulation based on the observation that often the measured effects of cooling inefficiencies are not located near the original source of the heat; in other words, heat may travel several meters through the data center before arriving at a temperature sensor. In both cases, our algorithms achieve the theoretical heat distribution recommendations, given discrete power states imposed by real-world constraints. We show how these algorithms can nearly halve cooling costs over the worst-case placement for a simple data center, and achieve an additional 18% in cooling savings beyond previous work. Based on these improvements we can eliminate more than 25% of the total cooling costs. Such savings in the 30,000 ft^2 data center mentioned earlier translate to a \$1 – \$2 million annual cost reduction. Furthermore, our work is complementary to current approaches; given a fixed cooling configuration, we quantify the cost of adding load to specific servers. A data center owner can use these costs to maximize the utilization per Watt of their compute and cooling infrastructure.

The rest of this paper is organized as follows. Section 2 elaborates the motivation for this work and discusses the limitations of conventional facilities-only approaches. Section 3 describes the goals of temperature-aware workload placement and discusses the algorithms that we propose — ZBD and MINHR — as well as three baseline algorithms provided for comparison. Sections 4 and 5 present our results and discuss their implications. Section 6 concludes the paper.

2 Motivation

As yesterday's clusters grow into today's data centers, infrastructure traditionally maintained by a facilities management team — such as cooling and the room's power grid — are becoming an integral part of data center design. No longer can data center operators focus solely on IT-level performance considerations, such as selecting the appropriate interconnect fiber or amount of memory per node. They now need to additionally evaluate issues dealing with power consumption and heat extraction.

For example, current-generation 1U servers consume over 350 Watts at peak utilization, releasing much of this energy as heat; a standard 42U rack of such servers consumes over 8 kW. Barroso et al estimate that the power density of the Google data center is three to ten times that of typical commercial data centers [10]. Their data center uses commodity mid-range servers; that density is likely to be higher with newer, more power-hungry server choices. As data centers migrate to bladed servers over the next few years, these numbers could potentially increase to 55 kW per rack [21].

2.1 Thermal Management Benefits

A thermal management policy that considers facilities components, such as CRAC units and the physical layout of the data center, and temperature-aware IT components, can:

Decrease cooling costs. In a 30,000 ft^2 data center with 1000 standard computing racks, each consuming 10 kW, the initial cost of purchasing and installing the computer room air conditioning (CRAC) units is \$2 – \$5 million; with an average electricity cost of \$100/MWhr, the annual costs for cooling alone are \$4 – \$8 million [23]. A data center that can run the same computational workload and cooling configuration, but maintain an ambient room temperature that is 5°C cooler, through intelligent thermal management can lower CRAC power consumption by 20% – 40% for a \$1 – \$3 million savings in annual cooling costs.

Increase hardware reliability. A recent study [28] indicated that in order to avoid thermal redlining, a typical server needs to have the air temperature at its front inlets be in the range of 20°C – 30°C. Every 10°C increase over 21°C decreases the reliability of long-term electronics by 50%. Other studies show that a 15°C rise increases hard disk drive failure rates by a factor of two [6, 13].

Decrease response times to transients and emergencies. Data center conditions can change rapidly. Sharp transient spikes in server utilization [7, 16] or the failure of a CRAC unit can upset the current environment in a matter of minutes or even seconds. With aggressive heat densities in the data center, such events can result in potentially disruptive downtimes due to the slow response times possible with the mechanical components at the facilities level.

Increase compaction and improve operational efficiencies. A high ratio of cooling power to compute power limits the compaction and consolidation possible in data centers, correspondingly increasing the management costs.

2.2 Existing Approaches

Data centers seek to provision the cooling adequately to extract the heat produced by servers, switches, and other hardware. Current approaches to data centers cooling provisioning are done at the facilities level. Typically, a data center operator will add the nameplate power ratings of all the servers in the data center — often with some additional slack for risk tolerance — and design a cooling infrastructure based on that number. This can lead to an excessive, inefficient cooling solution. This problem is exacerbated by the fact that the compute infrastructure in most data centers are provisioned for the peak (bursty) load requirement. It is estimated that typical operations of the data center often use only a fraction of the servers,

leading to overall low server utilization [18]. The compounded overprovisioning of compute and cooling infrastructure drives up initial and recurring costs. For every Watt of power consumed by the compute infrastructure, a modern data center expends another one-half to one Watt to power the cooling infrastructure [23, 28].

In addition, the granularity of control provided in current cooling solutions makes it difficult to identify and eliminate the specific sources of cooling inefficiencies. Air flow within a data center is complex, nonintuitive, and easy to disrupt [23]. Changes to the heating system — servers and other hardware — or the CRAC units will take minutes to propagate through the room, complicating the process of characterizing air flow within the room.

Past work on data center thermal management falls into one of two categories. First, optimizing the flow of hot and cold air in the data center. Second, minimizing global power consumption and heat generation. The former approaches evaluate layout of the computing equipment in the data center to minimize air flow inefficiencies (e.g., hot aisles and cold aisles) [28] or design intelligent system controllers to improve cold air delivery [23]. The latter approaches focus on location-oblivious, global system power consumption (total heat load) through the use of global power management [12, 25], load balancing [11, 24], and power reduction features in individual servers [14].

2.3 Temperature-aware Workload Placement

However, these approaches do not address the potential benefits from controlling the workload (and hence heat placement) from the point of view of minimizing the cooling costs. Addressing thermal and power issues at the IT level — by incorporating temperature-related metrics in provisioning and assignment decisions — is complementary to existing solutions. The last few years have seen a push to treat energy as a first-class resource in hardware and operating system design, from low-power processors to OS schedulers [29, 31]. A facilities-aware IT component operates at a finer granularity than CRAC units. It can not only react to the heat servers generate, but control when and where the heat arrives. During normal operations, a temperature-aware IT component can maintain an efficient thermal profile within the data center, resulting in reduced annual cooling costs. In the event of a thermal emergency, IT-level actions include scaling back on server CPU utilization, scaling CPU voltages [14], migrating or shifting workload [22, 11], and performing a clean shutdown of selected servers.

Figure 1 presents an informal sketch to illustrate the potential of this approach. The cooling costs of a data center are plotted as a function of the data center utilization — increased utilization produces larger heat loads, resulting

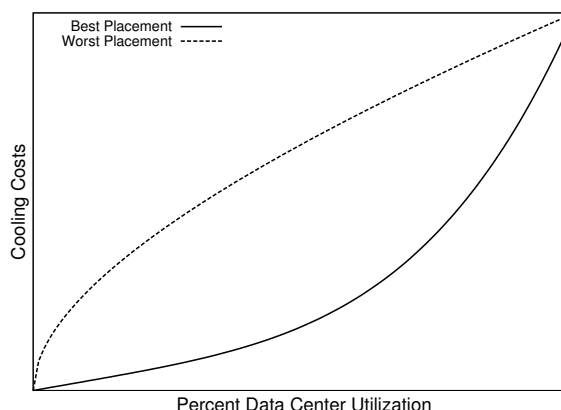


Figure 1: Approximate trends in cooling costs as a data center’s utilization increases. Workload placement algorithms affect cooling costs by the assignment choices they make. At the extreme ends — all servers idle and all servers used — there are no choices. However, at all other times there exists a best and a worst workload placement strategy.

in higher cooling costs. At any given data center utilization, there is a best and worst workload placement strategy. The difference between the two lines indicate the potential benefits from our approach.

As Figure 1 indicates, the benefits of our approach are limited at the two end points — a data center at “0%” utilization or at “100%” utilization does not offer much scope for workload placement to reduce cooling costs. In the former, all servers are idle; in the latter, all servers are in use. In neither case do we have any choice in how to deploy workload. The benefits from temperature-aware workload placement exist at intermediate utilization levels when we can choose how we place our workload. Typical data centers do not maintain 100% utilization for extended periods of time, instead operating at mid-level utilizations where we can leverage temperature-aware workload placement algorithms [18].

The slope and “knee” of each curve is different for each data center, and reflects the quality of the physical layout of the data center. For example, a “best placement” curve with a knee at high utilization indicates a well laid-out data center with good air flow. However, given the inefficiencies resulting from the coarse granularity of control in pure facilities-based approach, we expect most data centers to exhibit a significant difference between the worst-case and best-case curves.

3 Workload Placement Policies

At a high level, the goals of any temperature-aware workload placement policy are to

- Prevent server inlet temperatures from crossing a pre-defined “safe” threshold.
- Maximize the temperature of the air the CRAC units pump into the data center, increasing their operating efficiency.

This section provides a brief overview of the thermodynamics of cooling, how intelligent workload placement reduces CRAC unit power consumption, and describes our placement policies.

3.1 Thermodynamics

The cooling cycle of a typical data center operates in the following way. CRAC units operate by extracting heat from the data center and pumping cold air into the room, usually through a pressurized floor plenum. The pressure forces the cold air upward through vented tiles, entering the room in front of the hardware. Fans draw the cold air inward and through the server; hot air exits through the rear of the server. The hot air rises — sometimes with the aid of fans and a ceiling plenum — and is sucked back to the CRAC units. The CRAC units force the hot air past pipes containing cold air or water. The heat from the returning air transfers through the pipes to the cold substance. The now-heated substance leaves the room and goes to a chiller, and CRAC fans force the now-cold air back into the floor plenum.

The efficiency of this cycle depends on several factors, including the conductive substance and the air flow velocity, but is quantified by a *Coefficient of Performance* (COP). The COP is the ratio of heat removed (Q) to the amount of work necessary (W) to remove that heat:

$$\begin{aligned} COP &= \frac{Q}{W} \\ W &= \frac{Q}{COP} \end{aligned}$$

Therefore, the work necessary to remove heat is inversely proportional to the COP. A higher COP indicates a more efficient process, requiring less work to remove a constant amount of heat. For example, a cooling cycle with a COP of two will consume 50 kW to remove 100 kW of heat, whereas a cycle with a COP of five will consume 20 kW to remove 100 kW.

However, the COP for a cooling cycle is not constant, increasing with the temperature of the air the CRAC unit pushes into the plenum. We achieve cost savings by raising the plenum supply temperature, moving the CRAC units into a more efficient operating range. Figure 2 shows how the COP increases with higher supply temperatures for a typical water-chilled CRAC unit; this curve is from a water-chilled CRAC unit in the HP Utility Data Center. For example, if air returns to the CRAC unit at 20°C and

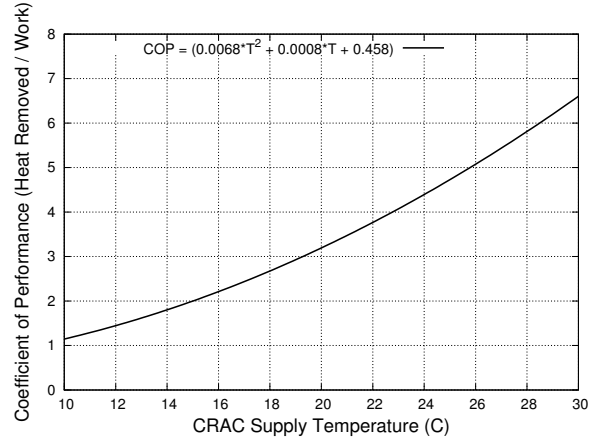


Figure 2: The Coefficient of Performance (COP) curve for the chilled-water CRAC units at the HP Labs Utility Data Center. As the target temperature of the air the CRAC pumps into the floor plenum increases, the COP increases, and the CRAC expends less energy to remove the same amount of heat.

we remove 10 kW of heat, cooling that air to 15°C, we expend 5.26 kW. However, if we raise the plenum supply temperature to 20°C, everything in the data center warms by 5°C. Cooling the same volume of air, now returning at 25°C, to 20°C removes the same 10 kW of heat, but only expends 3.23 kW. This is a power savings of almost 40%.

Consequently, our scheduling policies attempt to maximize cooling efficiency by raising the maximum temperature of the air coming from the CRAC units and flowing into the plenum. Obviously, this has to be done in a manner that maintains prevents the server inlet temperatures from crossing their redlining thermal threshold.

3.2 Terminology

At a fundamental level, we categorize power allocation algorithms as either *analog* or *digital*. “Analog” algorithms specify per-server power budgets from the continuous range of real numbers $[P_{off}, P_{max}]$. While analog algorithms provide a detailed per-server budget, they are hard to implement in practice. It may be possible to meet these goals — a data center operator may deploy fine-grained load balancing in a web farm [8], utilize CPU voltage scaling [14], or leverage virtual machines [1, 9] for batch workloads — but in practice it is difficult to meet and maintain precise targets for power consumption.

“Digital” algorithms assign one of several pre-determined discrete power states to each server. They select which machines should be off, idle, or in use, particularly for workloads that fully utilize the processors. They could also leverage the detailed relationship between server utilization and power consumption to allow few discrete utilization states. Additionally, a *well-ordered* digital al-

gorithm will create a list of servers sorted by their “desirability”; the list ordering is fixed for a given cooling configuration, but does not change for different data center utilization levels. Therefore, if data center utilization jumps from 50% to 60%, the servers selected for use at 50% are a proper subset of those selected at 60% utilization. Well-ordered algorithms simplify the process of integrating cooling-aware features with existing components such as SGE [4] or LSF [3], allowing us to use common mechanisms such as scheduling priorities. For example, SGE allows the administrator to define arbitrary “consumable” resources and simple formulas to force the scheduler to consider these resources when performing workload placement; modifying these resource settings is only necessary after a calibration run.

In this paper, we focus on algorithms that address the problem of discrete power states. We specifically focus on compute-intensive batch jobs such as multimedia rendering workloads, simulations, or distributed computation run for several hours [5]. These jobs tend to use all available CPU on a server, transforming the per-server power budgets available to a data center scheduler from a continuous range of $[P_{off}, P_{max}]$ to a discrete set of power states: $\{P_{off}, P_{idle}, P_1, \dots, P_N\}$, where P_j is the power consumed by a server fully utilizing j CPUs. Additionally, they also provide sufficient time for the thermal conditions in the room to reach steady-state. If additional power states are considered, Section 5 discusses how our algorithms scale in a straightforward manner.

3.3 Baseline Algorithms

We use three reference algorithms as a basis for comparison.

UniformWorkload and CoolestInlets

The first algorithm is UNIFORMWORKLOAD, an “intuitive” analog algorithm that calculates the total power consumed by the data center and distributes it evenly to each of the servers. We chose this algorithm because, over time, an algorithm that places workload randomly will approach the behavior of UNIFORMWORKLOAD. Each server in our data center consumes 150 Watts when idle and 285 Watts when at peak utilization. Thus, a 40% UNIFORMWORKLOAD will place $((285 - 150) \cdot 0.40) + 150 = 204$ Watts on each server.

The second baseline algorithm is COOLESTINLETS, a digital algorithm that sorts the list of unused servers by their inlet temperatures. This intuitive policy simply places workload on servers in the coldest part of the data center. Such an algorithm is trivial to deploy, given an instrumentation infrastructure that reports current server temperatures.

OnePassAnalog

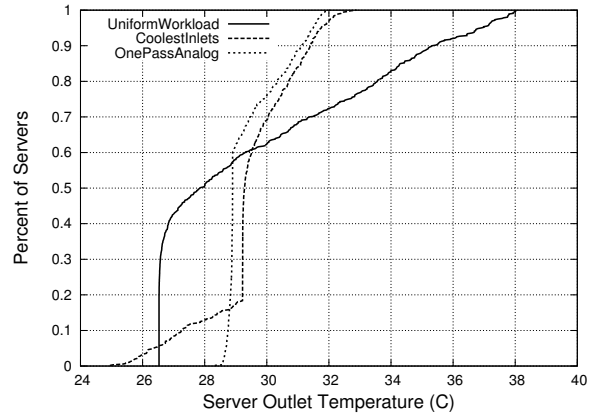


Figure 3: CDF of server exhaust temperatures for the three reference workload placement algorithms at 60% utilization. Both COOLESTINLETS and ONEPASSANALOG base workload placement decisions on data center conditions. However, ONEPASSANALOG has the least variance in server exhaust temperatures (4°C) leading to fewer heat buildups in the data center. Less variance allows us to raise CRAC supply temperatures further, increasing the COP, without causing thermal redlining.

The last policy is ONEPASSANALOG, an analog reprovisioning algorithm based on the theoretical thermodynamic formulation by Sharma et al [27], modified with the help of the original authors to allocate power on a per-server basis. The algorithm works by assigning power budgets in a way that attempts to create a uniform exhaust profile, avoiding the formation of any heat imbalances or “hot spots”. A data center administrator runs one calibration phase, in which they place a uniform workload on each server and observe each server’s inlet temperature. The administrator selects a reference $\{ \text{power, outlet temperature} \}$ tuple, $\{P_{ref}, T_{ref}^{out}\}$; this reference point can be one server, or the average server power consumption and outlet temperature within a row or throughout the data center. With this tuple, we calculate the power budget for each server:

$$P_i = \frac{T_{ref}^{out}}{T_i^{out}} \cdot P_{ref}$$

A server’s power budget, P_i , is inversely proportional to its outlet temperature, T_i^{out} . Intuitively, we want to add heat to cool areas and remove it from warm areas.

It is important to note that ONEPASSANALOG responds to heat buildup by changing the power budget at the location of the observed increase. Intuitively, this is similar to other approaches — including the motherboard’s thermal kill switch — in that it addresses the observed effect rather than the cause.

Figure 3 shows the CDF of server exhaust temperatures for the three reference workload placement algorithms in

```

ZONEBASEDDISCRETIZATION( $n, V, H, \alpha$ ) {
  while selected less than  $n$  servers {
    Get  $S_i$ , idle server with max power budget
     $P_{need} = P_{run} - P_{S_i}$ 
     $Weight_{Neighbors} = \alpha \cdot \text{size}(V) + \text{size}(H)$ 
     $P_{share} = P_{need} / Weight_{Neighbors}$ 
    Poach  $P_{share}$  from each of the  $H$  horizontal neighbors,
      ( $\alpha \cdot P_{share}$ ) from each of the  $V$  vertical neighbors
  }
}

```

Figure 4: The core of the ZBD algorithm. n is the number of servers we want, V is the set of neighbors along the vertical axis, H is the set of neighbors along the horizontal axis, and α is the ratio of power borrowed per-vertical to power borrowed per-horizontal. P_{run} is the amount of power necessary to run one server at 100% utilization; P_{S_i} is the amount of power the ONEPASSANALOG algorithm allocates to server i . In general, $P_{run} \geq P_{S_i}$.

a data center at 60% utilization. A data center that employs ONEPASSANALOG scheduling has less variance in its server's exhaust temperatures; UNIFORMWORKLOAD and COOLESTINLETS have server exhaust temperatures that vary by as much as 9°C – 12°C, whereas ONEPASSANALOG varies by less than 4°C; this indicates fewer localized “hot spots” and heat imbalances.

3.4 Zone-Based Discretization (ZBD)

Our first approach is based on the theoretical formulation behind ONEPASSANALOG [27]. This formulation assigns heat inversely proportional to the server's inlet temperature. However, it suffers from the drawback that it is analog; it does not factor in the specific discrete power states of current servers: $\{P_{idle}, \dots, P_N\}$. Therefore, the challenge is to discretize the recommended analog distribution to the available discrete power states. Our research showed that conventional discretization approaches — ones that are agnostic to the notion of heat distribution and transfer — that simply minimize the absolute error, can result in worse cooling costs.

The key contribution of ZBD is that, in addition to minimizing the discretization error over the entire data center, it minimizes the differences between its power distribution and ONEPASSANALOG at coarse granularities, or geographic *zones*.

ZBD chooses servers by using the notions of proximity-based heat distributions and *poaching*. When selecting a server on which to place workload, the chosen server borrows, or “poaches” power from its zone of immediate neighbors whose power budget is not already committed.

Within these two-dimensional zones, the heat produced by ZBD is similar to that produced by ONEPASSANALOG. Therefore, ZBD is an effective discretization of ONEPASSANALOG by explicitly capturing the underlying goal of ONEPASSANALOG: creating a uniform exhaust profile that reduces localized hot spots. A discretization approach that does not take this goal into account loses the benefits of ONEPASSANALOG.

Figure 4 describes the core of the ZBD discretization algorithm. ZBD allows us to define a variable-sized set of neighbors along the horizontal and vertical axes — H and V — and α , the ratio of power taken from the vertical to horizontal directions. These parameters enable us to mimic the physics of heat flow, as heat is more likely to rise than move horizontally. Consequently, “realistic” poaching runs set α larger than zero, borrowing more heavily vertically from servers in their rack.

Table 1 shows the operation of ZBD at a micro level, borrowing power from four vertical and two horizontal neighbors, giving the center server enough of a power budget to operate. The total amount of power and heat within the fifteen-server group remains the same, only shifted around slightly.

3.5 Minimizing Heat Recirculation (MinHR)

Our second approach is a new power provisioning policy that minimizes the amount of heat that recirculates within a data center: MINHR. Heat recirculation occurs for several reasons. For example, if there is not enough cold air coming up from the floor, a server fan can suck in air from other sources, usually hot air from over the top or around the side of racks. Similarly, if the air conditioning units do not pull the hot air back to the return vents or if there are obstructions to the air flow, hot air will mix with the incoming cold air supply. In all these cases, heat recirculation leads to increases in cooling energy.

Interestingly, some of these recirculation effects can lead to situations where the observed consequence of the inefficiency is spatially uncorrelated with its cause; in other words, the heat vented by one machine may travel several meters before arriving at the inlet of another server. We assert that an algorithm that minimizes hot air recirculation at the data center level will lead to lower cooling costs. Unlike ZBD, which reacts to inefficiencies by lowering the power budget at the site where heat recirculation is observed, MINHR focuses on the *cause* of inefficiencies. That is, it may not know how to lower the inlet temperature on a given server, but it will lower the total amount of heat that recirculates within the data center.

Therefore, unlike ZBD, we make no effort to create a uniform exhaust profile. The goals are to

- minimize the total amount heat that recirculates be-

184.61	216.77	207.15
184.44	216.80	207.41
186.24	216.88	207.66
189.25	216.86	207.82
193.41	216.82	207.89

(a) ONEPASSANALOG budgets.

184.61	216.77	207.15
184.44	216.80	207.41
186.24	216.88	207.66
189.25	216.86	207.82
193.41	216.82	207.89

(b) Select S_i and its neighbors.
 $P_{need} = 68.12$ Watts.

184.61	203.67	207.15
184.44	203.70	207.41
178.38	285.00	199.80
189.25	203.76	207.82
193.41	203.72	207.89

(c) Poach. $P_{share} = 7.86$ Watts,
 $(\alpha \cdot P_{share}) = 13.10$ Watts.

Table 1: The first iteration of ZBD with $n = 6$, $\text{size}(H) = 2$, $\text{size}(V) = 4$, and $\alpha = \frac{5}{3}$. The server with the highest power budget “poaches” power from its immediate neighbors. The total power allotted to these fifteen servers remains constant, but we now have a server with enough power to run at 100% utilization. At the end of this iteration, one server has enough power to run a full workload; after another $n - 1$ iterations, we will have selected our n servers.

fore returning to the CRAC units.

- maximize the power budget — and therefore the potential utilization — of each server.

First, we need a way to quantify the amount of hot air coming from a server or a group of servers that recirculates within the data center. We define δQ as

$$\delta Q = \sum_{i=1}^n C_p \cdot m_i \cdot (T_i^{in} - T_{sup})$$

Here, n is the number of servers in the data center, C_p is the specific heat of air (a thermodynamic constant measured with units of $\frac{W \cdot sec}{kg \cdot K}$), m_i is the mass flow of air through server i in $\frac{kg}{sec}$, T_i^{in} is the inlet temperature for

Pod	$\Delta \delta Q_j$	HRF_j	$\frac{HRF_j}{SRF}$	$Power_j$	δQ_j
1	1000	2	0.050	250	125
2	400	5	0.125	625	125
3	250	8	0.200	1000	125
4	80	25	0.625	3125	125

Table 2: Hypothetical MINHR calibration results and workload distribution for a 40U rack of servers divided into four pods of 10 servers each. ΔQ_{ref} during calibration is 2 kW; the final workload is 5 kW.

server i , and T_{sup} is the temperature of the cold air supplied by the CRAC units. In a data center with no heat recirculation — $\delta Q = 0$ — each T_i^{in} will equal T_{sup} .

Our workload placement algorithm will distribute power relative to the ratio of heat produced to heat recirculated:

$$P_i \propto \frac{Q_i}{\delta Q_i}$$

We run a two-phase experiment to obtain the heat recirculation data. This experiment requires an idle data center, but it is necessary to perform this calibration experiment once and only when there are significant changes to the hardware within the data center; for example, after the data center owner adds a new CRAC unit or adds new racks of servers. The first phase has the data center run a reference workload that generates a given amount of heat, Q_{ref} ; we also measure δQ_{ref} , the amount of heat recirculating in the data center. For the sake of simplicity, our reference state has each server idle.

The second phase is a set of sequential experiments that measure the heat recirculation of groups of servers. We bin the servers into *pods*, where each pod contains s adjacent servers; pods do not overlap. We define pods instead of individual servers to minimize calibration time and to ensure that each calibration experiment generates enough heat to create a measurable effect on temperature sensors in the data center. In each experiment, we take the next pod, j , and maximize the CPU utilization of all its servers simultaneously, increasing the total data center power consumption and heat recirculation. After the new data center power load and resulting heat distribution stabilize, we measure the new amount of heat generated, Q_j , and heat recirculating, δQ_j . With these, we calculate the *Heat Recirculation Factor* (HRF) for that pod, where

$$\begin{aligned} HRF_j &= \frac{Q_j - Q_{ref}}{\delta Q_j - \delta Q_{ref}} \\ &= \frac{\Delta Q_j}{\Delta \delta Q_j} \end{aligned}$$

Once we have the ratio for each pod, we use them to distribute power within the data center. We sum the HRF

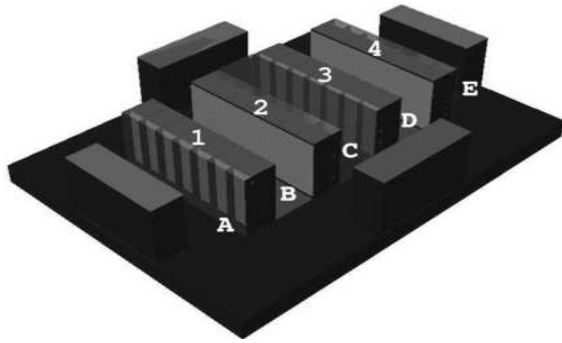


Figure 5: Layout of the data center. The data center contains 1120 servers in 28 racks, arranged in four rows of seven racks. The racks are arranged in a standard hot-aisle/cold-aisle configuration [28]. Four CRAC units push cold air into a floor plenum, which then enters the room through floor vents in aisles *B* and *D*. Servers eject hot air into aisles *A*, *C*, and *E*.

from each pod to get the *Summed Recirculation Factor* (SRF). To calculate the per-pod power distributions, we simply multiply the total power load by that pod’s *HRF*, divided by the *SRF*. This power budget distribution satisfies both of our stated goals; we maximize the power budget of each pod — maximizing the number of pods with enough power to run a workload — while minimizing the total heat recirculation within the data center. With this power distribution, each pod will recirculate the same amount of heat.

As before, we need to discretize the analog recommendations based on the *HRF* values for the power states in the servers. The scheduler then allocates workloads based on the discretized distribution. Note that the computed *HRF* is a property of the data center and is independent of load.

Table 2 shows an example of MINHR for a 40U rack of 1U servers divided into four pods. The resulting power budgets leads to identical amounts of heat from each pod recirculating within the data center. Although we could budget more power for the bottom pod to further minimize heat recirculation, but that would reduce the power budgets for other pods and lessen the number of available servers. Additionally, it is likely that the bottom pod has enough power to run all 10 servers at 100% utilization; increasing its budget serves no purpose, and instead reduces the amount of power available to other servers.

4 Results

This section presents the cooling costs associated with each workload placement algorithm.

4.1 Data Center Model

Given the difficulties of running our experiments on a large, available data center, we used Flovent [2], a Computational Fluid Dynamics (CFD) simulator, to model workload placement algorithms and cooling costs of the medium-sized data center shown in Figure 5. This methodology has been validated in prior studies [27].

The data center contains four rows with seven 40U racks each, for a total of 28 racks containing 1120 servers. The data center has alternating “hot” and “cold” aisles. The cold aisles, *B* and *D*, have vented floor tiles that direct cold air upward towards the server inlets. The servers eject hot air into the remaining aisles: *A*, *C*, and *E*. The data center also contains four CRAC units, each having the COP curve depicted in Figure 2. Each CRAC pushes air chilled to 15°C into the plenum at a rate of 10,000 $\frac{ft^3}{min}$. The CRAC fans consume 10 kW each.

The servers are HP Proliant DL360 G3s; each 1U DL360 has a measured power consumption of 150W when idle and 285W with both CPUs at 100% utilization. The total power consumed and heat generated by the data center is 168 kW while idle and 319.2 kW at full utilization. Percent utilization is measured as the number of machines that are running a workload. For example, when 672 of the 1120 servers are using both their CPUs at 100% and the other 448 are idle, the data center is at 60% utilization. To save time configuring each simulation, we modeled each pair of DL360s as a 2U server that consumed 300W while idle and 570W while at 100% utilization.

Calculating Cooling Costs

At the conclusion of each simulation, Flovent provides the inlet and exhaust temperature for each object in the data center. We calculate the cooling costs for each run based on a maximum safe server inlet temperature, T_{safe}^{in} , of 25°C, and the maximum observed server inlet temperature, T_{max}^{in} . We adjust the CRAC supply temperature, T_{sup} , by T_{adj} , where

$$T_{adj} = T_{safe}^{in} - T_{max}^{in}$$

If T_{adj} is negative, it indicates that a server inlet exceeds our maximum safe temperature. In response, we need to lower T_{sup} to bring the servers back below the system redline level.

Our cooling costs can be calculated as

$$C = \frac{Q}{COP(T = T_{sup} + T_{adj})} + P_{fan}$$

where Q is the amount of power the servers consume, $COP(T = T_{sup} + T_{adj})$ is our COP at $T_{sup} + T_{adj}$,

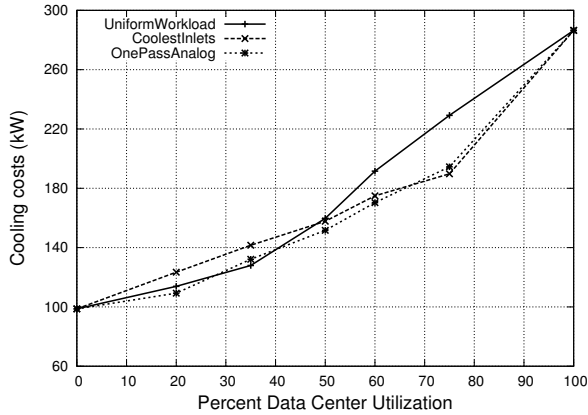


Figure 6: ONEPASSANALOG is consistently low, indicating a potential “best” cooling curve described in Figure 1. UNIFORMWORKLOAD performs well at low utilizations, but lacks the ability to react to changing conditions at higher utilizations. COOLESTINLETS performs well at higher utilizations, but is more expensive at low-range and mid-range utilization.

calculated from the curve in Figure 2, and P_{fan} is the total power consumed by the CRAC fans. Currently we assume a uniform T_{sup} from each CRAC due to the complications introduced by non-uniform cold air supply; we discuss these complications, proposed solutions, and ongoing work in Section 5.

4.2 Baseline Algorithms

Figure 6 shows the cooling costs for our three baseline algorithms. UNIFORMWORKLOAD performs well at low utilization by not placing excessive workload on servers that it shouldn’t. At high utilization, though, it places workload on all servers, regardless of the effect on cooling costs. In contrast, we see that ONEPASSANALOG performs well both at high and low data center utilization. It reacts well as utilization increases, scaling back the power budget on servers whose inlet temperatures increase. This avoids creating hot spots in difficult-to-cool portions of the room that would otherwise cause the CRAC units to operate less efficiently. COOLESTINLETS does well at high and mid-range utilization for this data center, but is about 10% more expensive than ONEPASSANALOG at low and moderate utilization.

4.3 ZBD

Parameter Selection

For ZBD to mimic the behavior of ONEPASSANALOG, we need to select parameters that reflect the underlying heat flow. Heat rises, so we set our α to be greater than 1, and our vertical neighborhood to be larger than our horizontal neighborhood. Our simulated servers are 2U high;

Zone Size	Avg Power	UW CoV	ZBD CoV
2U	462	0.009	0.008
4U	924	0.012	0.009
8U	1848	0.018	0.006
10U	2310	0.020	0.006

Table 3: Coefficient of variance (CoV) of differences in zonal power budgets between ONEPASSANALOG and the UNIFORMWORKLOAD (UW) and the ZBD algorithms at 60% utilization. Small coefficients indicate a distribution that mimics ONEPASSANALOG closely, creating a similar exhaust profile.

therefore our servers are 8.89cm (3.5in) tall and 60.96cm (24in) wide. Since heat intensity is inversely proportional to the square of the distance from the source, it makes little sense to poach two servers or more (greater than one meter) in either horizontal direction. Noting that our rows are 20 servers high and 7 across, we maintain this ratio both in poaching distance and poaching ratio. We set our vertical neighborhood to be three servers in either direction, and our α to $\frac{20}{7}$. These parameters are simple approximations; in section 5 we discuss methods of improving upon ZBD parameter selection.

Results

The next question is whether we met our goals of matching the high-level power allocation behavior of ONEPASSANALOG. In order to quantify the similarity of any two algorithms’ power distributions, we break each 40U rack into successively larger zones; zones are adjacent and do not overlap. We sum the servers’ power allocations to get that zone’s budget. Table 3 shows the per-pod variance between the ONEPASSANALOG zone budgets those of UNIFORMWORKLOAD and ZBD power distributions are to the ONEPASSANALOG power budgets at different granularities. Unsurprisingly, UNIFORMWORKLOAD has the largest variance at any zone size; it continues to allocate power to each server, regardless of room conditions. However, ZBD closely mirrors the power distribution budgeted by ONEPASSANALOG.

Figure 7 shows the relative costs of ZBD against our three baseline algorithms as we ramp up data center utilization. Like ONEPASSANALOG, ZBD performs well both at low and high utilizations. Most importantly, we see that ZBD mimics the behavior and resulting cooling costs of ONEPASSANALOG within two percent. Even with intuitive parameter selection and the challenge of discretizing the analog distribution, we met or exceeded the savings available using the theoretical best workload assignment algorithm from previously published work.

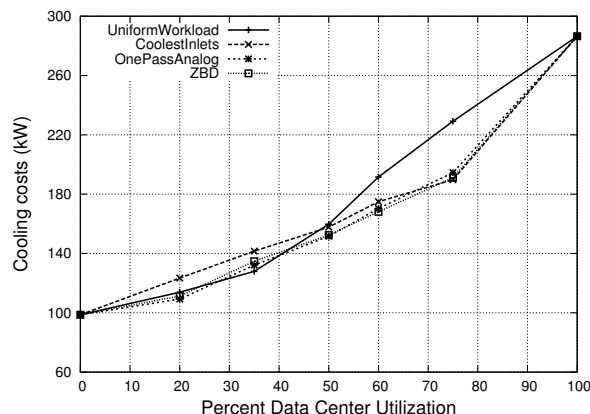


Figure 7: ZBD compared to our baseline algorithms. ZBD also works well at high and low utilizations, staying within $\pm 3\%$ of ONEPASSANALOG.

4.4 MinHR

Calibration

The performance of MINHR depends on the accuracy of our calibration experiments. Our goals in selecting calibration parameters for MINHR, such as pod sizes and our Q_{ref} , were to allow for a reasonable calibration time and a reasonable degree of accuracy. If pod sizes are too small, we may have too many pods and an unreasonably long real-world calibration time — approximately twenty minutes per pod — and the $\Delta\delta Q_i$ may be too small to create any observable change. Since calibration times using Flovent are significantly longer than in real life — one to two hours per pod — we chose a pod size of 10U. This translates to a 1.35 kW ΔQ_i , as we increase each server from 150W to 285W. While smaller pods may give us data at a finer granularity, the magnitude of δQ may be too small to give us an accurate picture of how that pod's heat affects the data center.

Figure 8 demonstrates the importance of locating the sources of heat recirculation. It shows the warmest 10% of server inlets for our calibration phase and for the recirculation workload at the top pod of a rack on the end of row 4. Even though we increase the total power consumption of the servers by only 0.80% (1.35 kW), the cooling costs increase by 7.56%. A large portion of the hot exhaust from these servers does not return to a CRAC unit, instead returning to the servers. Inlets at the top of row 4 increase by over 1°C , and servers at the same end of row 3 see an increase in inlet temperature of over $\frac{2}{3}^\circ\text{C}$.

With MINHR, unlike ONEPASSANALOG, it was not necessary to perform any form discretization on the analog power budgets from. Figure 9 shows the CDF of server power budgets while our data center is at 60% utilization. In ONEPASSANALOG, of the 1120 servers, only 84 fall

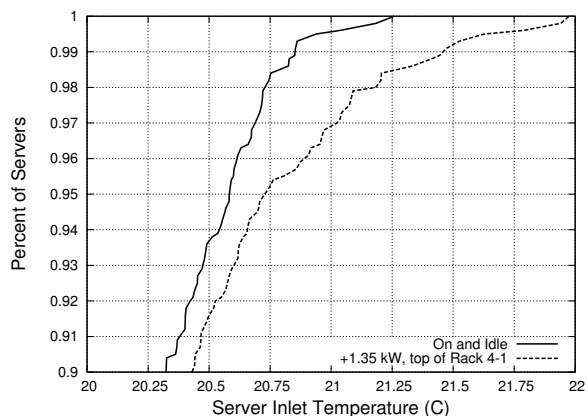


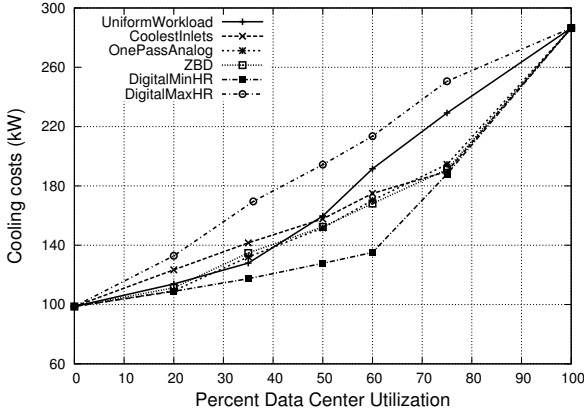
Figure 8: CDF of the warmest ten percent of server inlets for the MINHR phase-one calibration workload, and after adding a total of 1.35 kW to ten servers during a phase-two recirculation workload. A 1°C increase in the maximum server inlet temperature results in 10% higher cooling costs. This phase-two workload was at the top corner of row 4.

outside the operating range of our DL360s, thus necessitating the use of ZBD.

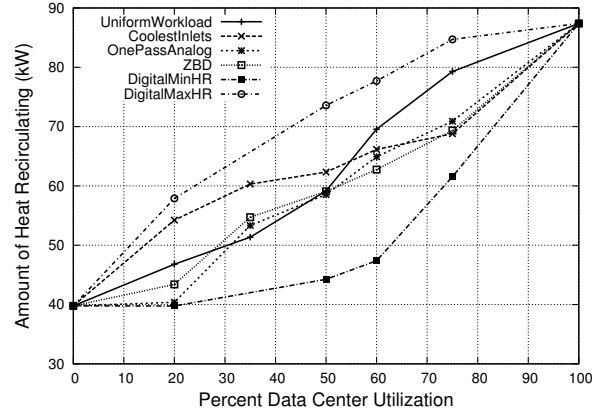
However, MINHR assigns power budgets between 13 and 3876 Watts per 1U server, with only 160 falling within the operating range; we chose simply to sort the servers by their power budget and chose the $X\%$ with the highest budgets, where X is our target utilization. We define ANALOGMINHR as the original, unrealistic power distribution, and the sort-and-choose as DIGITALMINHR. For the sake of clarity, we define DIGITALMAXHR as DIGITALMINHR in reverse; we start at the bottom of the list, using the worst candidates and moving up.

Results

Figure 10(a) compares our four previous algorithms against DIGITALMINHR and DIGITALMAXHR. At mid-range utilization, DIGITALMINHR saves 20% over ONEPASSANALOG, 30% over UNIFORMWORKLOAD, and nearly 40% over DIGITALMAXHR. The costs of each algorithm are related to the heat recirculation behaviors they cause. At low utilization, DIGITALMAXHR quickly chooses servers whose exhaust recirculates extensively, whereas DIGITALMINHR does not save much over ONEPASSANALOG; this indicates that initially ONEPASSANALOG also minimizes heat recirculation. As utilization increases, however, all algorithms except DIGITALMINHR end up placing load on servers that recirculate large amounts of heat; DIGITALMINHR knows exactly which servers to avoid. At near-peak utilizations, however, DIGITALMINHR has run out of “good” servers to use, driving up cooling costs.



(a) Cooling costs for our baseline algorithms, ZBD, and best and worst heat-recirculation-based algorithms.



(b) The amount of heat recirculating. Note that the increase in heat recirculation closely mirrors the increase in cooling costs.

Figure 10: At mid-range utilizations, DIGITALMINHR costs 20% less than ONEPASSANALOG, 30% less than UNIFORMWORKLOAD and almost 40% less than the worst possible workload distribution.

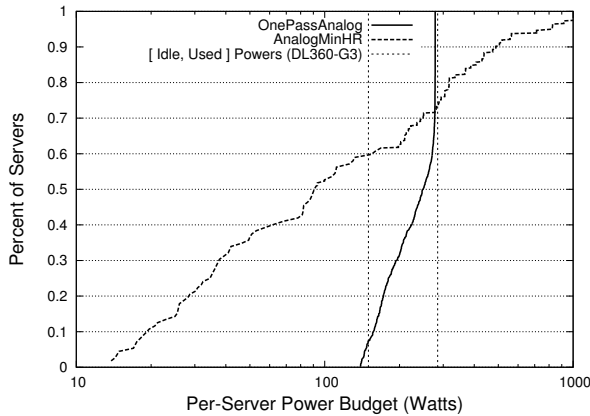


Figure 9: CDF of ONEPASSANALOG and ANALOGMINHR budgets at 60% utilization. ONEPASSANALOG budgets fall within the DL360's operating range; this facilitates ZBD's zone-based discretization. The minimum and maximum ANALOGMINHR budgets are more than an order of magnitude outside this range, eliminating the need for or effectiveness of any discretization algorithm.

Figure 10(b) graphs δQ for each algorithm. DIGITALMINHR achieves its goal, minimizing recirculation and cooling costs until there are no “good” servers available. Conversely, DIGITALMAXHR immediately chooses “bad” servers, increasing power consumption by 30.2 kW and heat recirculation by 18.1 kW. Note that cooling costs are closely related to the amount of heat recirculating within the data center.

4.5 ZBD and MinHR Comparison

At a glance, DIGITALMINHR provides significant savings over all other workload placement algorithms. It directly addresses the cause of data center cooling inefficiencies, and is constrained only by the physical air flow design of the data center. Unfortunately, the calibration phase is significantly longer than the one ZBD requires. A real-world calibration of our model data center would take 56 hours; this is not unreasonable, as the entire calibration run would complete between Friday evening and Monday morning. However, a calibration run is necessary whenever the physical layout of the room changes, or after a hardware or cooling upgrade. Conversely, ZBD is consistent and the best “reactive” digital algorithm. It only requires one calibration experiment; for our data center, this experiment would complete within a half-hour.

Ultimately, the data center owner must decide between long calibration times and savings in cooling costs. If the cooling configuration or physical layout of the data center will not change often, then a MINHR-based workload placement strategy yields significant savings.

5 Discussion

Additional Power States

Our previous experiment assumes the computer infrastructure only had two power states: idle and used. However, many data center management infrastructure components — such as networked power switches, blade control planes, and Wake-On-LAN-enabled Ethernet cards — al-

# Off	Power (kW)	Cooling (kW)	% Savings
56	273.0	156.9	16.54
112	264.6	142.9	23.96
168	256.2	134.4	28.51
224	247.8	126.00	32.96

Table 4: We leverage MINHR’s sorted list of server “desirability” to select servers to turn off during 75% utilization. We reduce the power consumed by the computer infrastructure by 12%, yet reduce cooling costs by nearly one-third.

low us to consider “off” as another power state. Both the algorithms can leverage additional power states to allow them to more closely match the analog power budgets.

To demonstrate the potential for increased improvements, we focus on some experiments using the best algorithm from the last section. DIGITALMINHR’s per-pod *HRF* values allow us to sort servers by heat recirculation and power down or fully turn off the “worst” servers. Table 4 presents the results of turning 5, 10, and 15% of the “worst” servers off during 75% utilization while using the DIGITALMINHR placement algorithm. Initially the computer infrastructure was consuming 281.4 kW, and expending 187.9 kW to remove this heat. Turning off only 56 servers, 8.4 kW of compute power, reduces cooling costs by nearly one-sixth. MINHR with an “off” option reduces cooling costs by nearly another third by turning off 20% of the servers.

When compared to the savings achieved by ONEPASSANALOG over UNIFORMWORKLOAD, this approach represents a factor of three increase in those cooling savings, reducing UNIFORMWORKLOAD cooling costs by nearly 60%. These long-term savings may be reduced, however, by the decreased hardware reliability caused by power-cycling servers.

How far to perfection?

In this section, we compare our results to the absolute theoretical minimum cost of heat removal, as defined by physics. It is possible to calculate the absolute minimum cooling costs possible, given the COP curve of our CRAC units. Assume we formulate the perfect workload placement algorithm, one that eliminates hot air recirculation. In that case, we have the situation described in Section 3.5: CRAC supply temperatures equal the maximum safe server inlet temperatures. Plugging the data from the COP curve in figure 2, we obtain $W_{optimal} = \frac{Q}{4.728}$.

Figure 11 compares all our workload placement algorithms against the absolute minimum costs, as governed by the above equation. It should be noted that the absolute minimum represents a realistically unobtainable point as is evident from the benefits it can obtain even at the 100%

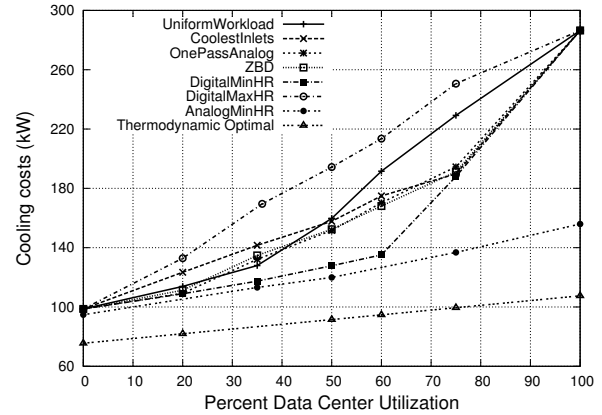


Figure 11: Cooling costs for all workload placement algorithms, ANALOGMINHR, and the absolute minimum costs for our data center.

data point where there is no slack in workload placement. However, in spite of this, for our simple data center at mid-range workloads, DIGITALMINHR achieves over half the possible savings as compared to UNIFORMWORKLOAD. These savings are through changes a data center administrator can make entirely at the IT level in software, such as modifying a batch queue or other server assignment scheduler. Furthermore, as discussed earlier, these changes are complementary to other facilities approaches, including improved rack locations and cooling configurations.

Instrumentation and Dynamic Control

The work discussed in this paper assumes an instrumentation infrastructure in the data center such as Splice [19] that provides current temperature and power readings to our algorithms. Related work in the data instrumentation space includes PIER [15], Ganglia [26] and Astro-labe [30]. Additionally, algorithms such as ONEPASSANALOG, ZBD, and MINHR include calibration phases based on past history. These phases could potentially be sped by systematic thermal profile evaluations through a synthetic workload generation tool such as *sstress* [20]. A moderately-sized data center of 1000 nodes will take about two days to calibrate fully. At the end of the calibration phase, however, which we will have the power budgets for that data center. These budgets are constant unless the cooling or computational configuration changes, such as by adding or removing servers.

Further, the work discussed in this paper pertains to static workload assignment in a batch scheduler to reduce cooling costs from a heat distribution perspective. We assume that the cooling configuration is not being optimized concurrently; in other words, CRAC units may not vary their supply temperatures individually, or change their fan

speeds at all. However, some data centers exist where aggressive cooling optimizations could concurrently vary the cooling configurations.

For these scenarios, we are currently exploring the possibility of using system identification techniques from control theory [17] to “learn” how the thermal profile of the data center changes as cooling settings change. These identification tools will reveal the relationships between cooling parameters and heat recirculation observations, allowing us to expand the uses of temperature-aware workload placement to include such features as emergency actions in the event of CRAC unit failure. For the time being, however, a data center owner could perform one calibration phase with each CRAC unit off to simulate the failure of that unit and obtain the relative power budgets and server ordering.

6 Conclusion

Cooling and heat management are fast becoming the key limiters for emerging data center environments. As data centers grow during the foreseeable future, we must expand our understanding of cooling technology and how to apply this knowledge to data center design from an IT perspective. In this paper, we explore temperature-aware resource provisioning to control heat placement from a systems perspective to reduce cooling costs.

We explore the physics of heat transfer, and present methods for integrating it into batch schedulers. To capture the complex thermodynamic behavior in the data center, we use simple heuristics that use information from steady-state temperature distribution and simple cause-effect experiments to calibrate sources of inefficiencies. To capture the constraints imposed by real-world discrete power states, we propose location-aware discretization heuristics that capture the notion of zonal heat distribution, as well as recirculation-based placement. Our results show that these algorithms can be very effective in reducing cooling costs. Our best algorithm nearly halves cooling costs when compared to the worst-case scenario, and represents a 165% increase in the savings available through previously published methods. All these savings are obtained purely in software without any additional capital costs. Furthermore, our results show that these improvements can be larger with more aggressive use of power states, as is likely in future systems.

Though we focus mainly on cooling costs in this paper, our algorithms can also be applied to other scenarios such as graceful degradation under thermal emergencies. In these cases, compared to longer timescales associated with the more mechanical-driven facilities control, temperature-aware workload placement can significantly improve the response to failures and emergencies. Simi-

larly, the principles underlying our heuristics can be leveraged in the context of more complex dynamic control algorithms as well.

In summary, as future data centers evolve to include ever larger number of servers operating in increasingly denser configurations, it will become critical to have heat management solutions that go beyond conventional cooling optimizations at the facilities level. We believe that approaches like ours that straddle the facilities and systems management boundaries to holistically optimize for power, heat, and cooling, will be an integral part of future data center solutions to address these challenges.

A Acknowledgments

We would like to thank Tzi-cker Chiueh, our shepherd, and the anonymous reviewers for their comments and suggestions. Janet Wiener provided invaluable aid in refining ZBD. Keith Farkas has contributed throughout the research and development process as we work to deploy our work in a live data center. We would also like to thank Chandrakant Patel, Cullen Bash, and Monem Beitelmal for their assistance with all things cooling.

Special thanks to Rocky Shih, our Flovent guru.

References

- [1] VMware – Virtual Computing Environments. <http://www.vmware.com/>.
- [2] Flovent version 2.1, Flometrics Ltd, 81 Bridge Road, Hampton Court, Surrey, KT8 9HH, England, 1999.
- [3] LSF Scheduler from Platform Computing, October 2004. <http://www.platform.com/products/LSF/>.
- [4] Sun Grid Engine, October 2004. <http://www.sun.com/software/gridware/>.
- [5] The Seti @ Home Project, October 2004. <http://setiathome2.ssl.berkeley.edu/>.
- [6] D. Anderson, J. Dykes, and E. Riedel. More Than an Interface—SCSI vs. ATA. In *Proceedings of the 2nd Usenix Conference on File and Storage Technologies (FAST)*, San Francisco, CA, March 2003.
- [7] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. Technical Report HPL-1999-35R1, HP Research Labs, September 1999.
- [8] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-Aware Request Distribution in Cluster-Based Network Servers. In *Proceedings of the USENIX 2000 Technical Conference*, 2000.
- [9] P. Barham, B. Dragovic, K. Faser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles*, Bolton Landing, New York, October 2003.

- [10] L. A. Barroso, J. Dean, and U. Holzle. Web Search for a Planet: The Google Cluster Architecture. In *IEE Micro*, pages 22–28, March–April 2003.
- [11] D. J. Bradley, R. E. Harper, and S. W. Hunter. Workload-based Power Management for Parallel Computer Systems. *IBM Journal of Research and Development*, 47:703–718, 2003.
- [12] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 103–116, October 2001.
- [13] G. Cole. Estimating Drive Reliability in Desktop Computers and Consumer Electronics. In *Technology Paper TP-338.1, Seagate Technology*, November 2000.
- [14] K. Flautner and T. Mudge. Vertigo: Automatic Performance-Setting for Linux. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 105–116, Boston, Massachusetts, December 2002. ACM Press.
- [15] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, September 2003.
- [16] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *Proceedings of the 2002 International World Wide Web Conference*, pages 252–262, May 2002.
- [17] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance Isolation and Differentiation for Storage Systems. In *Proceedings of the Twelfth International Workshop on Quality of Service*, pages 67–74, June 2004.
- [18] J. D. Mitchell-Jackson. Energy Needs in an Internet Economy: A Closer Look at Data Centers. Master's thesis, University of California, Berkeley, 2001.
- [19] J. Moore, J. Chase, K. Farkas, and P. Ranganathan. A Sense of Place: Toward a Location-aware Information Plane for Data Centers. In *Hewlett Packard Technical Report TR2004-27*, 2004.
- [20] J. Moore, J. Chase, K. Farkas, and P. Ranganathan. Data Center Workload Monitoring, Analysis, and Emulation. In *Eighth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2005.
- [21] J. Mouton. Enabling the vision: Leading the architecture of the future. In *Keynote speech, Server Blade Summit*, 2004.
- [22] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 361–376, Boston, Massachusetts, December 2002.
- [23] C. D. Patel, C. E. Bash, R. Sharma, and M. Beitelmal. Smart Cooling of Data Centers. In *Proceedings of the Pacific RIM/ASME International Electronics Packaging Technical Conference and Exhibition (IPACK03)*, July 2003.
- [24] E. Pinheiro, R. Bianchini, E. Carrera, and T. Heath. Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, September 2001.
- [25] K. Rajamani and C. Lefurgy. On Evaluating Request-Distribution Schemes for Saving Energy in Server Clusters. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, March 2003.
- [26] F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler. Wide Area Cluster Monitoring with Ganglia. In *Proceedings of the IEEE Cluster 2003 Conference*, Hong Kong, 2003.
- [27] R. K. Sharma, C. L. Bash, C. D. Patel, R. J. Friedrich, and J. S. Chase. Balance of Power: Dynamic Thermal Management for Internet Data Centers. *IEEE Internet Computing*, 9(1):42–49, January 2005.
- [28] R. F. Sullivan. Alternating Cold and Hot Aisles Provides More Reliable Cooling for Server Farms. In *Uptime Institute*, 2000.
- [29] A. Vahdat, A. R. Lebeck, and C. S. Ellis. Every joule is precious: The case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.
- [30] R. van Renesse and K. P. Birman. Scalable Management and Data Mining using Astrolabe. In *Proceedings for the 1st International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.
- [31] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing Energy as a First Class Operating System Resource. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, October 2002.

CHAMELEON: a self-evolving, fully-adaptive resource arbitrator for storage systems

Sandeep Uttamchandani[†] Li Yin[‡] Guillermo A. Alvarez[†]
John Palmer[†] Gul Agha^{*}

[†] *IBM Almaden Research Center*

[‡] *University of California, Berkeley*

^{*} *University of Illinois at Urbana-Champaign*

{sandeepu,alvarezg,jdp}@us.ibm.com, yinli@eecs.berkeley.edu, agha@cs.uiuc.edu

Abstract

Enterprise applications typically depend on guaranteed performance from the storage subsystem, lest they fail. However, unregulated competition is unlikely to result in a fair, predictable apportioning of resources. Given that widespread access protocols and scheduling policies are largely best-effort, the problem of providing performance guarantees on a shared system is a very difficult one. Clients typically lack accurate information on the storage system's capabilities and on the access patterns of the workloads using it, thereby compounding the problem. CHAMELEON is an adaptive arbitrator for shared storage resources; it relies on a combination of self-refining models and constrained optimization to provide performance guarantees to clients. This process depends on minimal information from clients, and is fully adaptive; decisions are based on device and workload models automatically inferred, and continuously refined, at runtime. Corrective actions taken by CHAMELEON are only as radical as warranted by the current degree of knowledge about the system's behavior. In our experiments on a real storage system CHAMELEON identified, analyzed, and corrected performance violations in 3-14 minutes—which compares very favorably with the time a human administrator would have needed. Our learning-based paradigm is a most promising way of deploying large-scale storage systems that service variable workloads on an ever-changing mix of device types.

1 Introduction

A typical consolidated storage system at the multi-petabyte level serves the needs of inde-

pendent, paying customers (e.g., a storage service provider) or divisions within the same organization (e.g., a corporate data center). Consolidation has proven to be an effective remedy for the low utilizations that plague storage systems [10], for the expense of employing scarce system administrators, and for the dispersion of related data into unconnected islands of storage. However, the ensuing resource contention makes it more difficult to guarantee a portion of the shared resources to each client, regardless of whether other clients over- or under-utilize their allocations—guarantees required by the prevalent *utility* model.

This paper addresses the problem of allocating resources in a fully automated, cost-efficient way so that most clients experience predictable performance in their accesses to a shared, large-scale storage utility. Hardware costs play a dwindling role relative to managing costs in current enterprise systems [10]. Static provisioning approaches are far from optimal, given the high burstiness of I/O workloads and the inadequate available knowledge about storage device capabilities. Furthermore, efficient static allocations do not contemplate hardware failures, load surges, and workload variations; system administrators must currently deal with those by hand, as part of a slow and error-prone observe-act-analyze loop. Prevalent access protocols (e.g., SCSI and FibreChannel) and resource scheduling policies are largely best-effort; unregulated competition is unlikely to result in a fair, predictable resource allocation.

Previous work on this problem includes management policies encoded as sets of rules [13, 27], heuristic-based scheduling of individual I/Os [7, 12, 15, 19], decisions based purely on feedback loops [17, 18] and on the predictions of

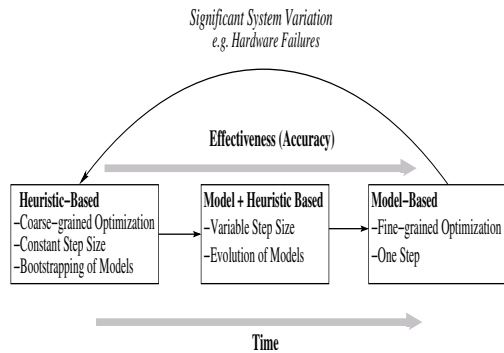


Figure 1: CHAMELEON moves along the line according to the quality of the predictions generated by the internally-built models at each point in time.

models for system components [1, 2, 3]. The resulting solutions are either not adaptive at all (as in the case of rules), or dependent on models that are costly to develop, or ignorant of the system’s performance characteristics as observed during its lifetime.

This paper’s main contribution is a novel technique for providing performance guarantees in shared storage systems, based on a combination of performance models, constrained optimization, and incremental feedback. CHAMELEON is a framework in which clients whose negotiated Service Level Agreement (SLAs) are not being met get access to additional resources freed up by *throttling* (i.e., rate-limiting) [7, 17] competing clients. Our goal is to make more accurate throttling decisions as we learn more about the characteristics of the running system, and of the workloads being presented to it. As shown in Figure 1, CHAMELEON operates at any point in a continuum between decisions made based on relatively uninformed, deployment-independent heuristics on the one hand, and blind obedience to models of the particular system being managed on the other hand.

CHAMELEON can react to workload changes in a nimble manner, resulting in a marginal number of quality of service (QoS) violations. In our experiments on a real storage system using real-world workload traces, CHAMELEON managed to find the set of throttling decisions that yielded the maximum value of the optimization function, while minimizing the amount of throttling required to meet the targets and while maximizing the number of clients whose QoS requirements are satisfied. Since our approach does not depend on pre-existing device or workload models, it can

be easily deployed on heterogeneous, large-scale storage systems about which little is known. Our ultimate vision, of which CHAMELEON is just a part, is to apply a variety of corrective actions to solve a variety of systems management problems while operating on incomplete information [25].

Section 2 presents the architecture of CHAMELEON. We then proceed to describe the main components: the models (Section 3), the reasoning engine (Section 4), the base heuristics (Section 5), and the feedback-based throttling executor (Section 6). Section 7 describes our prototype and experimental results. Section 8 reviews previous research in the field; we conclude in Section 9.

2 Overview of CHAMELEON

CHAMELEON is a framework for providing predictable performance to multiple clients accessing a common storage infrastructure, as shown in Figure 2. Multiple hosts connect to storage devices in the *back end* via interconnection fabrics. CHAMELEON can monitor, and optionally delay, every I/O processed by the system; this can be implemented at each host (as in our prototype), or at logical volume managers, or at block-level virtualization appliances [9]. Each workload j has a known SLA_j associated with it, and uses a fixed set of components—its *invocation path*—such as controllers, logical volumes, switches, and logical units (LUNs). When SLAs are not being met, CHAMELEON identifies and throttles workloads; when it detects unused bandwidth, it unthrottles some of the previously-throttled workloads.

Our SLAs are *conditional*: a workload will be guaranteed a specified upper bound on average I/O latency, as long as its I/O rate (i.e., the throughput) is below a specified limit. An SLA is *violated* if the rate is below the limit, but latency exceeds its upper bound. If workloads exceed their stated limits on throughput, the system is under no obligation of guaranteeing any latency. Obviously, such rogue workloads are prime candidates for throttling; but in some extreme cases, well-behaved workloads may also need to be restricted. CHAMELEON periodically evaluates the SLAs, i.e., the average latency and throughput value of each workload; depending on how much the workload is being throttled, it receives tokens (one per I/O) for flow control using a leaky bucket protocol [24]. The periodic interval for SLA evaluation has to be large enough to smooth out bursty intervals, and small enough for

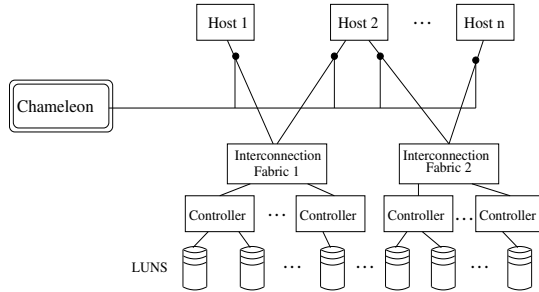


Figure 2: System model.

the system to be reasonably responsive; we empirically set this interval to 60 s. in our prototype implementation.

The core of CHAMELEON consists of four parts, as shown in Figure 3:

- **Knowledge base:** by taking periodic performance samples on the running system, CHAMELEON builds internal *black-box models* of system behavior without any human supervision. Models get better as time goes by, for CHAMELEON refines them automatically.
- **Reasoning engine:** CHAMELEON employs optimization techniques, informed by the black-box models. It computes throttle values, and quantifies the statistical confidence of its own decisions.
- **Designer-defined policies:** As a fallback mechanism, we maintain a set of fixed heuristics specified by the system designer for system-independent, coarse-grained resource arbitration.
- **Informed feedback module:** The general guiding principle is to take radical corrective action as long as that is warranted by the available knowledge about the system. If the confidence value from the solver is below a certain threshold (e.g., during bootstrapping of the models), CHAMELEON falls back on the fixed policies to make decisions.

3 Knowledge base

CHAMELEON builds models in an automatic, unsupervised way. It uses them to characterize the capabilities of components of the storage system, the workload being presented to them, and its expected response to different levels of throttling.

Models based on simulation or emulation require a fairly detailed knowledge of the system's internals; analytical models require less, but device-specific optimizations must still be taken into account to obtain accurate predictions [26]. Black-box models are built by recording and correlating inputs and outputs to the system in diverse states, without regarding its internal structure. We chose them because of properties not provided by the other modeling approaches: black-box models make very few assumptions about the phenomena being modeled, and can readily evolve when it changes. Because of this, they are an ideal building block for an adaptive, deployment-independent management framework that doesn't depend on pre-existing model libraries.

At the same time, the black-box models used in CHAMELEON are less accurate than their analytical counterparts; our adaptive feedback loop compensates for that. The focus of this paper is to demonstrate how several building blocks can work together in a hybrid management paradigm; we do not intend to construct good models, but to show that simple modeling techniques are adequate for the problem. CHAMELEON's models are constructed using Support Vector Machines (SVM) [16], a machine-learning technique for regression. This is similar to the CART [29] techniques for modeling storage device performance, where the response of the system is measured in different system states and represented as a best-fit curve function. Table-based models [2], where system states are exhaustively recorded in a table and used for interpolation, are not a viable solution as they represent the model as a very large lookup table instead of the analytic expressions that our constraint solver takes as input.

Black-box models depend on collecting extensive amounts of performance samples. Some of those metrics can be monitored from client hosts, while others are tallied by each component—and collected via proprietary interfaces for data collection, or via standard protocols such as SMI-S [20].

A key challenge is bootstrapping, i.e., how to make decisions when models have not yet been refined. There are several solutions for this: run a battery of tests in non-production mode to generate baseline models, or run in a monitor-only mode until models are sufficiently refined, or start from a pre-packaged library (e.g., a convenient oversimplification such as an M/M/1 queueing system.) We follow different approaches for dif-

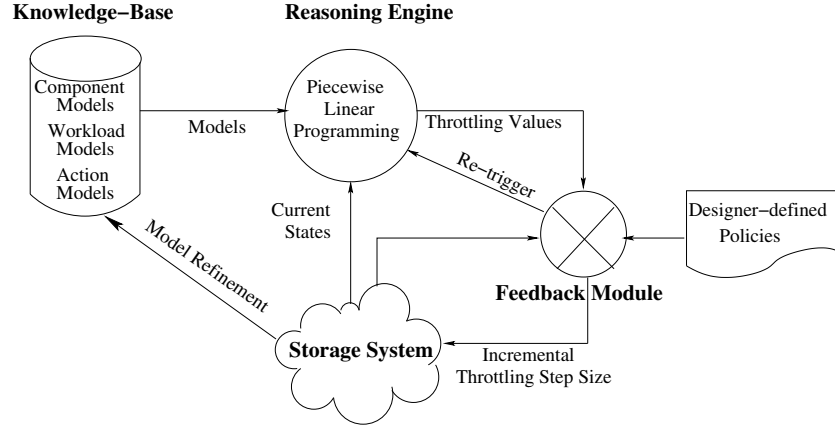


Figure 3: Architecture of CHAMELEON.

ferent model types.

3.1 Component models

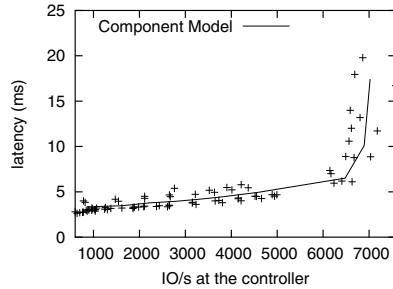


Figure 4: Component model.

A component model predicts values of a delivery metric as a function of workload characteristics. CHAMELEON can in principle accommodate models for any system component. In particular, the model for the response time of a storage device i takes the form: $c_i(req_size, req_rate, rw_ratio, random/sequential, cache_hit_rate)$. Function c_i is inherently non-linear, but can be approximated as piecewise linear with a few regions; a projection of a sample c_i is shown in Figure 4. Another source of error is the effect of multiple workloads sending interleaved requests to the same component. We approximate this nontrivial computation by estimating the wait time for each individual stream as in a multi-class queueing model [14]; more precise solutions [5] incorporate additional workload characteristics. The effects of caching at multiple levels (e.g., hosts, virtualization engines, disk array controllers, disks)

also amplify errors.

We took the liberty of bootstrapping component models by running off-line calibration tests against the component in question: a single, unchanging, synthetic I/O stream at a time, as part of a coarse traversal of c_i 's parameter space.

3.2 Workload models

Representation and creation of workload models has been an active area of research [6]. In CHAMELEON, workload models predict the load on each component as a function of the request rate that each workload injects into the system. For example, we denote the predicted rate of requests at component i originated by workload j as $w_{i,j}(workload_request_rate_j)$. In real scenarios, function $w_{i,j}$ changes continuously as workload j changes or other workloads change their access patterns (e.g., a workload with good temporal locality will push other workloads off the cache). To account for these effects, we represent function $w_{i,j}$ as a *moving average* [23] that gets recomputed by SVM every n sampling periods.

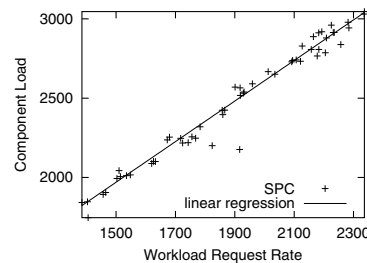


Figure 5: Workload model for SPC.

Figure 5 shows the workload models for the

SPC web-search trace [21] accessing a 24-drive RAID 1 LUN on an IBM FASTT 900 storage controller. From the graph, a workload request rate of 1500 IOPS in SPC translates to 2000 IOPS at the controller.

In practical systems, reliable workload data can only be gathered from production runs. We therefore bootstrap workload models by collecting performance observations; CHAMELEON resorts to throttling heuristics in the meantime, until workload models become accurate enough.

3.3 Action models

In general, action models predict the effect of corrective actions on workload requirements. The throttling action model computes each workload's average request rate as a function of the token issue rate, i.e., $a_j(token_issue_rate_j)$. Real workloads exhibit significant variations in their I/O request rates due to burstiness and to ON/OFF behaviors [5]. We model a as a linear function: $a_j(token_issue_rate_j) = \theta \times token_issue_rate_j$ where $\theta = 1$ initially for bootstrapping. This simple model assumes that the components in the workload's invocation path are not saturated.

Function a_j will, in general, also deviate from our linear model because of performance-aware applications (that modify their access patterns depending on the I/O performance they experience) and of higher-level dependencies between applications that magnify the impact of throttling.

4 Reasoning engine

The reasoning engine computes the rate at which each workload stream should be allowed to issue I/Os to the storage system. It is implemented as a constraint solver (using piecewise-linear programming [8]) that analyzes all possible combinations of workload token rates and selects the one that optimizes an administrator-defined *objective function*, e.g., “minimize the number of workloads violating their SLA”, or “ensure that highest priority workloads always meet their guarantees”. Based on the errors associated with the models, the output of the constraint solver is assigned a *confidence value*.

It should be noted that the reasoning engine is not just invoked upon an SLA violation to decide throttle values, but also periodically to unthrottle the workloads if the load on the system is reduced.

4.1 Intuition

The reasoning engine relies on the component, workload, and action models as oracles on which to base its decision-making. Figure 6 illustrates a simplified version of how the constraint solver builds a candidate solution: 1) for each component used by the *underperforming* workload (i.e., the one not meeting its SLA), use the component's model to determine the change in request rate at the component required to achieve the needed decrease in component latency; 2) query the model for each workload using that components, to determine which change in the workload's I/O injection rate is needed to relieve the component's load; 3) using the action model, determine the change in the token issue rate needed for the sought change in injection rate; 4) record the value of the objective function for the candidate solution. Then repeat recursively for all combinations of component, victim workload, and token issue rates. The reasoning engine is actually more general: it considers *all* solutions, including the ones in which the desired effect is achieved by the combined results of throttling more than one workload.

4.2 Formalization in CHAMELEON

We formulate the task of computing throttle values in terms of variables, objective function, and constraints as follows.

Variables: One per workload, representing its token issue rate: t_1, t_2, \dots

Objective function: Workloads are pigeonholed into one of four regions as in Figure 7, according to their current request rate, latency, and SLA goals. Region names are mostly self-explanatory—*lucky* workloads are getting a higher throughput while meeting the latency goal, and *exceeded* workloads get higher throughput at the expense of high latency.

Many objective functions can be accommodated by the current CHAMELEON prototype (e.g., all linear functions); moreover, it is possible to switch them on the fly. For our experiments, we minimized

$$\sum_{i \notin \text{failed}} \left| P_{quad_i} P_{W_i} \frac{SLA_{W_i} - a_i(t_i)}{SLA_{W_i}} \right|$$

where P_{W_i} are the *workload priorities*, P_{quad_i} are the *quadrant priorities* (i.e., the probability that workloads in each region will be selected as throttling candidates), and $a_i(t_i)$ represents the

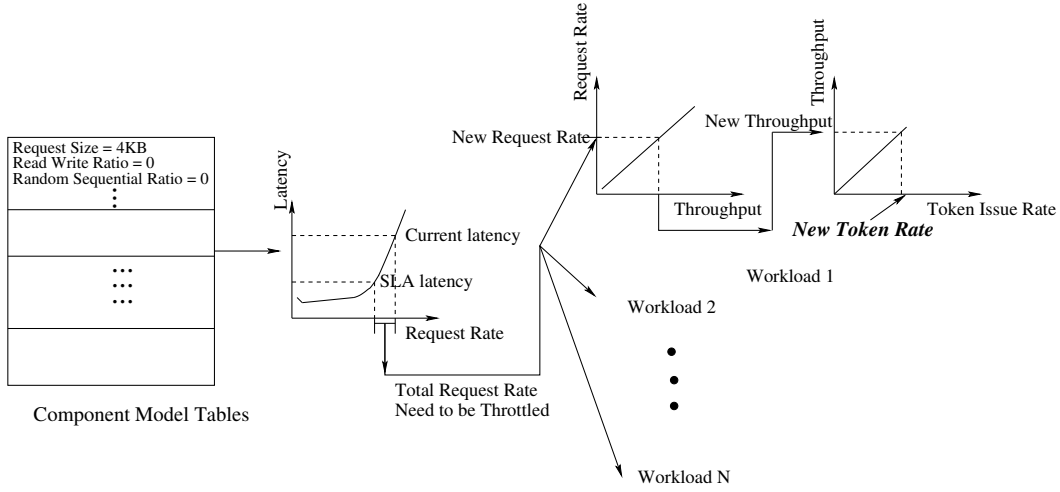


Figure 6: Overview of constrained optimization.

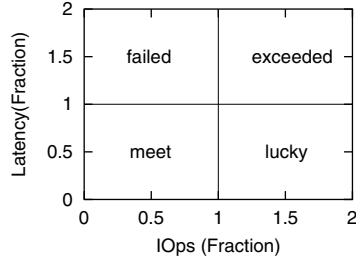


Figure 7: Workload classification. Region limits correspond to the 100% of the SLA values.

action model for W_i . Table 1 provides some insight into this particular choice.

Constraints: Constraints are represented as inequalities: the latency of a workload should be less than or equal to the value specified in the SLA. More precisely, we are only interested in solutions that satisfy $latency_{W_j} \leq SLA_{W_j}$ for all workloads W_j running in the system. We estimate the contribution of component i to the latency of W_j by composing our models in the knowledge base, i.e., $latency_{i,j} = c_i(w_{i,j}(a_j(t_j)))$.

For example, the latency constraint for a single workload W_1 running in the system with its I/O requests being served by a storage controller followed by physical disks is

$$c_{controller}(w_{controller,1}(a_1(t_1))) + c_{disks}(w_{disks,1}(a_1(t_1))) \leq SLA_1$$

In a more general example, workloads W_1, W_5 share the storage controller:

Intuition	How it is captured
The lower a workload's priority, the higher its probability of being throttled	The solver minimizes the objective function; violating the SLA of a higher priority workload will reflect as a higher value for $P_{W_i} \frac{SLA_{W_i} - a_i(t_i)}{SLA_{W_i}}$
Workloads in the lucky or exceeded region have a higher probability of being throttled	This is ensured by the P_{quad_i} variable in the objective function; it has higher values for lucky and exceeded (e.g., $P_{meet} = 1, P_{exceed} = 8, P_{lucky} = 32$). It is also possible to define P_{quad_i} as a function.
Workloads should operate close to the SLA boundary	By definition of the objective function; it is also possible to add a bimodal function, to penalize workloads operating beyond their SLA.

Table 1: Internals of the objective function.

$$total_req_{controller} = w_{controller,1}(a_1(t_1)) + w_{controller,5}(a_5(t_5));$$

$$total_req_{disks} = w_{disks,1}(a_1(t_1)) + w_{disks,5}(a_5(t_5));$$

$$c_{controller}(total_req_{controller}) + c_{disks}(total_req_{disks}) \leq SLA_1$$

4.3 Workload unthrottling

CHAMELEON invokes the reasoning engine periodically, to re-assess token issue rates; if the load on the system has decreased since the last invocation, some workloads may be unthrottled to re-distribute the unused resources based on workload priorities and average I/O rates. If a workload is consistently wasting tokens issued for it (because it has less significant needs), unused tokens will be considered for re-distribution; on the other hand, if the workload is using all its tokens, they won't be taken away from it, no matter how low its priority is. CHAMELEON makes unthrottling decisions using the same objective function with additional "lower-bound" constraints such as not allowing any I/O rate to become lower than its current average value.

4.4 Confidence on decisions

There are multiple ways of capturing statistical confidence values [14]. CHAMELEON uses the following formula to capture both the errors from regression and from residuals (i.e., models being used on inputs where they were not trained):

$$S_p = S \sqrt{1 + \frac{1}{n} + \frac{(x_p - \bar{x})^2}{\sum x^2 - n\bar{x}^2}}$$

where S is the standard error, n is the number of points used for regression, and \bar{x} is the mean value of the predictor variables used for regression. S_p represents the standard deviation of the predicted value y_p using input variable x_p . In CHAMELEON, we represent the confidence value CV of a model as the inverse of its S_p , and we define the overall confidence on the reasoning engine's decisions as $CV_{component} \times CV_{workload} \times CV_{action}$.

5 Designer-defined policies

The system designer defines heuristics for coarse-grained throttling control. Heuristics are used to make decisions whenever the predictions of the models cannot be relied upon—either during bootstrapping or after significant system changes such as hardware failures. Sample heuristics include "if system utilization is greater than 85%, start throttling workloads in the *lucky* region", or "if the workload-priority variance is less than 10%, uniformly throttle all workloads sharing the component".

These heuristics can be expressed in a variety of ways such as Event-Condition-Action (ECA) rules or hard-wired code. In any case, fully specifying corrective actions at design time is an error-prone solution to a highly complex problem [25], especially if they are to cover a useful fraction of the solution space and to accommodate priorities. It is also very hard to determine accurate threshold values to differentiate different scenarios, in the absence of any solid quantitative information about the system being built. In CHAMELEON, the designer-defined heuristics are implemented as simple hard-wired code which is a modified version of the throttling algorithm used in Sleds [7]:

1. Determine the *compList* of components being used by the underperforming workload.
2. For each component in the *compList*, add the non-underperforming workloads using it to the *candidateList*.
3. Sort the *candidateList* first by current operating quadrant: *lucky* first, then *exceed*, then *meet*. Within each quadrant, sort by workload priority.
4. Traverse the *candidateList* and throttle each workload, either uniformly or proportionally to its priority (the higher the priority, the less significant the throttling).

6 Informed feedback module

The feedback module (Figure 8) incrementally throttles workloads based on the decisions of either the reasoning engine or the system-designer heuristics. If CHAMELEON is following the reasoning engine, throttling is applied at incremental *steps* whose size is proportional to the confidence value of the constraint solver; otherwise, the throttling step is a small constant value.

After every m throttling steps, the feedback module analyzes the state of the system. If any of the following conditions is true, it re-invokes the reasoning engine; otherwise it continues applying the same throttling decisions in incremental steps:

- Latency increases for the underperforming workload (i.e., it moves away from the *meet* region).
- A non-underperforming workload moves from *meet* or *exceed* to *lucky*.
- Any workload undergoes a 2X or greater variation in the request rate or any other ac-

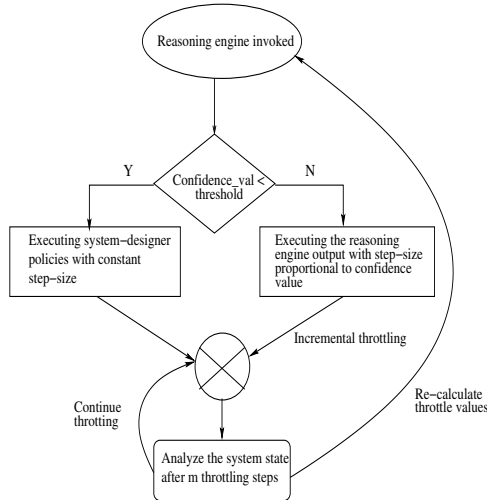


Figure 8: Operation of the feedback module.

cess characteristic, compared to the values at the beginning of throttling.

- There is a 2X or greater difference between predicted and observed response times for a component.

7 Experimental evaluation

The experimental setup consists of a host machine generating multiple I/O streams on a shared storage infrastructure. The host is an IBM x-series 440 server (2.4GHz 4-way Intel Pentium 4 with 4GB RAM running Redhat Server Linux, 2.1 kernel); the back-end storage is a 24-drive RAID 1 LUN created on a IBM FASTT 900 storage controller with 512MB of on-board NVRAM, and accessed as a raw device so that there is no I/O caching at the host. The host and the storage controller are connected using a 2Gbps FibreChannel (FC) link.

The key capability of CHAMELEON is to regulate resource load so that SLAs are achieved. The experimental results use numerous combinations of synthetic and real-world request streams to evaluate the effectiveness of CHAMELEON. As expected, synthetic workloads are easier to handle compared to their real-world counterparts that exhibit burstiness and highly variable access characteristics.

7.1 Using synthetic workloads

The synthetic workload specifications used in this section were derived from a study done in the

context of the Minerva project [1]. Since the workloads are relatively static and controlled, our action and workload models have small errors as quantified by the correlation coefficient r [14]. In general, the closer r is to 1, the more accurate the models are; in this experiment, the component model has $r = 0.72$, and the workload and action models have $r > 0.9$.

These tests serve two objectives. First, they evaluate the correctness of the decisions made by the constraint solver. Throttling decisions should take into account each workload's priority, its current operating point compared to the SLA, and the percentage of load on the components generated by the workload. Second, these tests quantify the effect of model errors on the output values of the constraint solver and how incremental feedback helps the system converge to an optimal state.

Workload	Request size [KB]	Rd/wrt ratio	Seq/rnd ratio	Foot-print [GB]
W_1	27.6	0.98	0.577	30
W_2	2	0.66	0.01	60
W_3	14.8	0.641	0.021	50
W_4	20	0.642	0.026	60

Table 2: Synthetic workload streams.

We report experimental results by showing each workload's original and new (i.e., post-throttling) position in the classification chart, as defined in Figure 7.

Case 1: Effect of workload and quadrant priorities

Figure 9 compares the direct output of the constraint solver with priority values for the workloads ($W_1 = 8, W_2 = 8, W_3 = 2, W_4 = 8$) and the SLA quadrant priorities (failed = 16, meet = 2, exceed = 8, lucky = 8). In comparison to the no priority scenario, W_3 and W_2 are throttled more when priorities are assigned for the workloads and quadrants respectively. This is because the constraint solver optimizes the objective function by throttling the lower priority workloads more aggressively before moving on to the higher priority ones.

Case 2: Usage of the component by the workload

This test is a sanity check with workload W_5 operating primarily from the controller cache (2KB

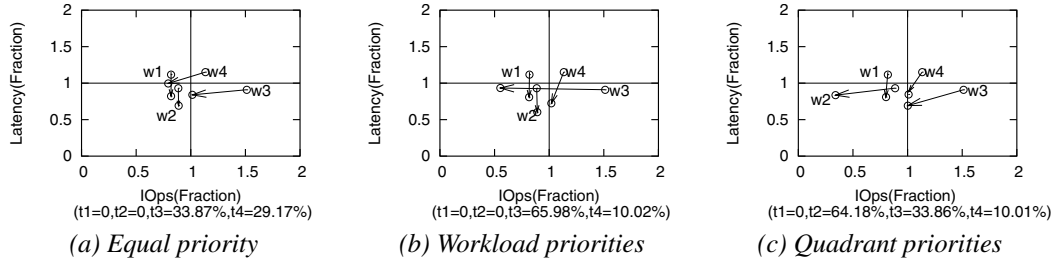


Figure 9: Effect of priority values on the output of the constraint solver.

sequential read requests). Because W_5 does not consume disk bandwidth, the reasoning engine should not attempt to solve the SLA violation for W_1 by throttling W_5 even if W_5 has the lowest priority. As shown in Figure 10, the reasoning engine selects W_2 and W_3 .

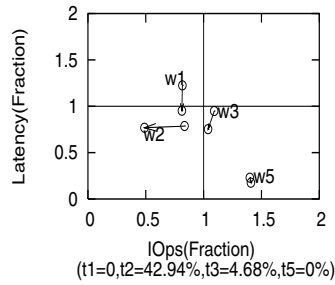


Figure 10: Sanity test for the reasoning engine (workload W_5 operating from controller cache.)

7.2 Replaying real-world traces

For these experiments, we replay the web-search *SPC* trace [21] and HP's *Cello96* trace [11]. Both are block-level traces with timestamps recorded for each I/O request. We use approximately 6 hours of *SPC* and one day of *Cello96*. To generate a reasonable I/O load for the storage infrastructure, *SPC* was replayed 40 times faster and *Cello96* was replayed 10 times faster.

In addition to the traces, we used a phased, synthetic workload was used; this workload was assigned the highest priority. In an uncontrolled case i.e. without throttling, with three workloads running on the system, one or more of them violate their SLA. Figure 11 shows the throughput and latency values for uncontrolled case. For all the figures in this subsection, there are four parts (ordered vertically): the first plot represents the throughput for the *SPC*, *Cello96*, and the synthetic workload. The second, third, and fourth

plots represent the latency for each of these workloads respectively.

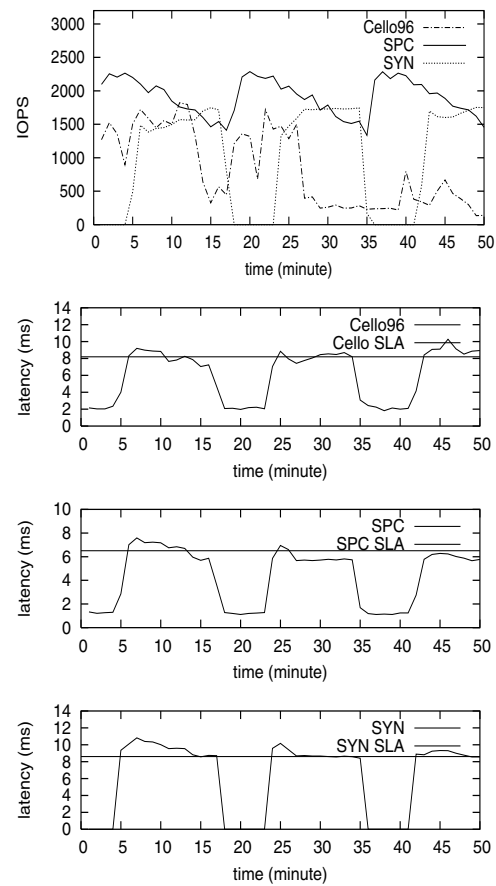


Figure 11: Uncontrolled throughput and latency values for real-world workload traces.

The aim of the tests is to evaluate the following:

- The throttling decisions made by CHAMELEON for converging the workloads towards their SLA.
- The reactivity of the system with throttling and periodic unthrottling of workloads

(under reduced system load).

- The handling of unpredictable variations in the system that cause errors in the model predictions, forcing CHAMELEON to use the sub-optimal but conservative designer-defined policies.

For these experiments, the models were reasonably accurate (component $r = 0.68$, workload $r = 0.7$, and action $r = 0.6$). In addition, the SLAs for each workload are: Cello96 1000 IOPS with 8.2ms latency, SPC 1500 IOPS with 6.5 ms latency and 1600 IOPS with 8.6ms latency for the synthetic workload unless otherwise specified.

Case 1: Solving SLA violations using throttling

The behavior of the system is shown in figure 12. To explain the working of CHAMELEON, we divide the time-series into phases (shown as dotted vertical line in the figures) described as follows:

Phase 0 ($t=0$ to $t=5$ min): Only the SPC and Cello96 traces are running on the system; the latency values of both these workloads are significantly below the SLA.

Phase 1 ($t=5$ min to $t=13$ min): The phased synthetic workload is introduced in the system. This causes an SLA violation for the Cello96 and synthetic traces. CHAMELEON triggers the throttling of the SPC and Cello96 workloads (Cello96 is also throttled because it is operating in the exceeded region, means it is sending more than it should. Therefore, it is throttled even if its SLA latency goal is not met.) The system uses a feedback approach to move along the direction of the output of the constraint solver. In this experiment, the feedback system starts from 30% of the throttling value and uses step size is 8% (30% and 8% are decided according to the confidence value of the models). It took the system 6 minutes to meet the SLA goal and stop the feedback.

Phase 2 ($t=13$ min to $t=20$ min): The system stabilizes after the throttling and all workloads can meet their SLAs.

Phase 3 ($t=20$ min to $t=25$ min): The synthetic workload enters the OFF phase. During this time, the load on the system is reduced, but the throughput of Cello96 and SPC remains the same.

Phase 4 (beyond $t=25$ min): The system is stable, with all the workloads meeting their SLAs. As a side note, around $t=39$ min the throughput

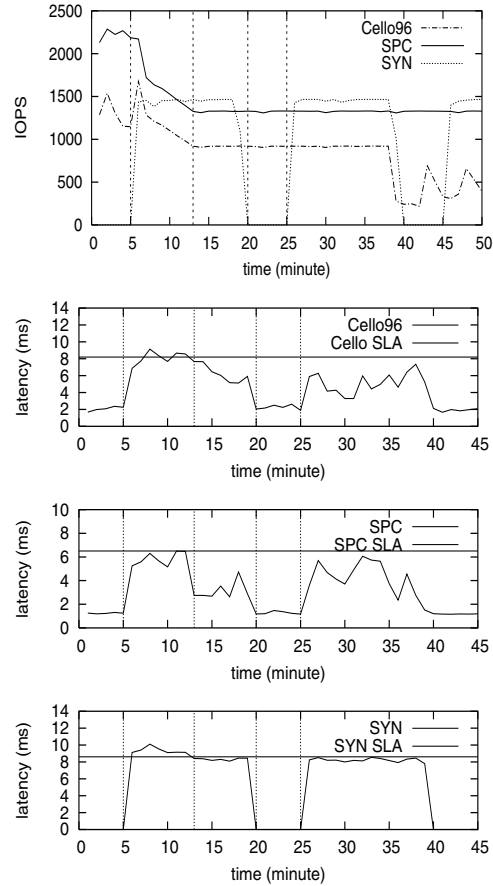


Figure 12: Throughput and latency values for real-world workload traces with throttling (without periodic unthrottling.)

of Cello96 decreases further; this is because of characteristics of the trace.

Figure 12 shows the effectiveness of the throttling: all workloads can meet their SLA after throttling. However, because the lack of an unthrottling scheme, throttled workloads have no means to increase their throughput even when tokens are released by other workloads. Therefore, the system is underutilized.

Case 2: Side-by-side throttling and unthrottling of workloads

The previous experiment demonstrates the effectiveness of throttling. Figure 13 shows throttling combined with unthrottling of workloads during reduced system load. Compared to Figure 12, the key differences are: 1) SPC and Cello96 increase their request-rate when the system load is reduced ($t=17$ min to $t=27$ min), improving overall system utilization; 2) the system has a non-

zero settling time when the synthetic workload is turned on ($t=27$ min to $t=29$ min). In summary, unthrottling allows for better system utilization, but requires a non-zero settling time for recovering the resources.

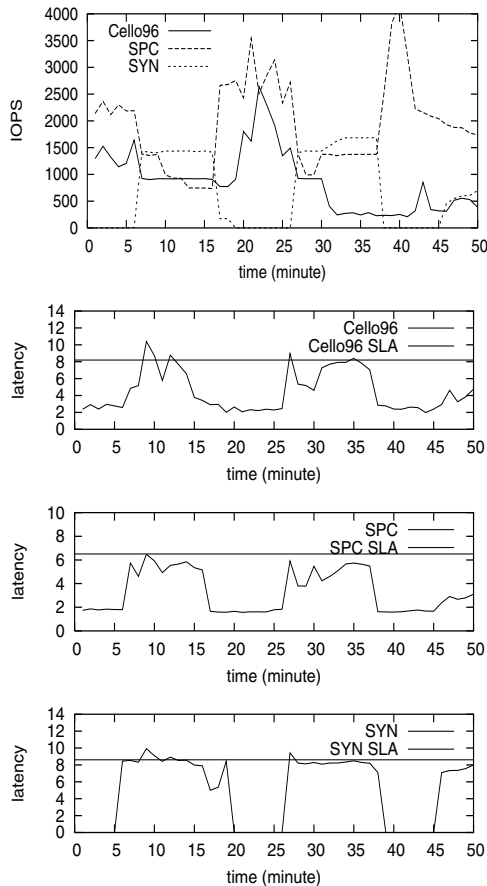


Figure 13: Throughput and latency values for real-world workload traces with throttling and periodic unthrottling.

Case 3: Handling changes in the confidence value of models at run-time

This test demonstrates how CHAMELEON handles changes in the confidence values of the models at run-time; these changes can be due to unpredictable system variations (hardware failures) or un-modeled properties of the system (such as changes in the workload access characteristics that change the workload models). It should be noted that refining the models to reflect the changes will not be instantaneous; in the meantime, CHAMELEON should have the ability to detect decreases in the confidence value and switch

to a conservative management mode (e.g., using designer-defined policies, or generate a warning for a human administrator).

Figure 14 show the reaction of the system when the access characteristics of the SPC and Cello96 workloads are synthetically changed such that the cache hit rate of Cello96 increases significantly (in reality, a similar scenario arise due to changes in the cache allocation to individual workload streams sharing the controller) and the SPC is doing more random access (sequential random ratio increases from 0.11 to 0.5). In the future, we plan to run experiments with hardware failures induced on the RAID 1 logical volume.

The SLAs used for this test are: Cello96 has a SLA with 1000 IOPS with 7ms latency, SPC is 2000 IOPS with 8.8ms latency and the synthetic workloads has a SLA with 1500 IOPS and 9ms latency.

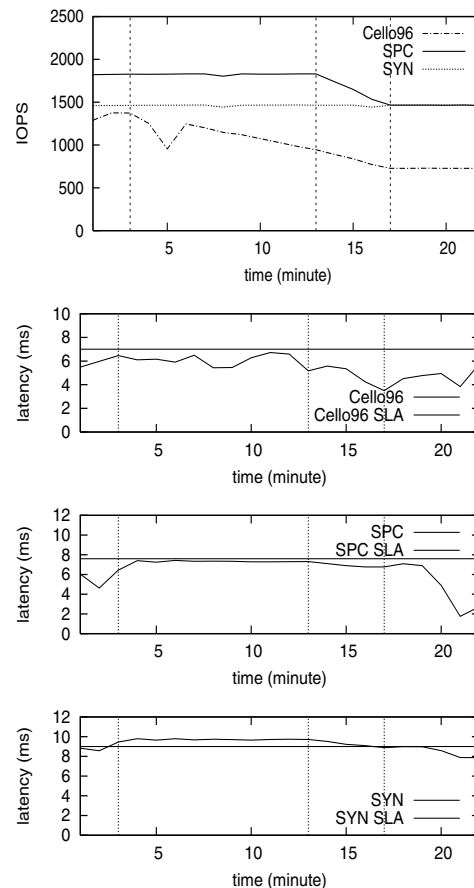


Figure 14: Handling a change in the confidence value of the models at run-time.

Phase 0 (at $t=3$ mins): The synthetic workload violates its latency SLA. In response,

CHAMELEON decides to throttle the Cello96 workload (using the original workload model). The output of the reasoning engine as a confidence value of 65%

Phase 1 ($t = 3$ min to $t = 13$ min): The feedback module continues to throttle for 3 consecutive increments; since the latency of the synthetic workload does not change, it re-invokes the reasoning engine. The output of the reasoning engine is similar to the previous invocation (since the models haven't changed), but its confidence value is lower (because of the higher differences between predicted and observed model values). This repeats for consecutive invocations of the reasoning engine after which the feedback module switches to use the designer-defined policies.

Phase 2 ($t = 13$ min to $t = 17$ min): A simple designer policy the CHAMELEON uses is to throttle all the non-violating workloads uniformly (*uniform pruning*). Both SPC and Cello96 are throttled in small steps (5% of their SLA IOPS) till the latency SLA of the synthetic workload is satisfied.

Phase 3 (beyond $t = 17$ min): All workloads are meeting their SLA goals and the system is stabilized.

8 Related work

Most storage management frameworks (including all commercial tools, e.g., BMC Patrol [4]) encode policies as ECA *rules* [27, 13] that fire when some precondition is satisfied—typically, when one or more system metrics cross predetermined thresholds. Rules are a clumsy, error-prone programming language; they front-load all the complexity into the work of creating them at design time, in exchange for simplicity of execution at run time. Administrators are expected to account for all relevant system states, to know which corrective action to take in each case, to specify useful values for all the thresholds that determine when rules will fire, and to make sure that the right rule will fire if preconditions overlap. Moreover, simple policy changes can translate into modifications to a large number of rules. Rule-based systems are only as good as the human who wrote the rules; they can just provide a coarse level of control over the system. Some variations rely on case-based reasoning [28] to iteratively refine rules from a *tabula rasa* initial knowledge base. This approach does not scale well to real systems, because of the exponential size of the search space that is explored in an un-

structured way. In contrast, CHAMELEON relies on constrained optimization to steer the search in the full space of throttle values, and uses its dynamically refined models in lieu of fixed thresholds.

Feedback-based approaches use a narrow window of the most recent performance samples to make allocation decisions based on the difference between the current and desired system states. They are not well-suited for decision-making with multiple variables [22], and can oscillate between local optima. Façade [19] controls the queue length at a single storage device; decreasing the queue length is equivalent to throttling the combination of all workloads, instead of (as in CHAMELEON) selectively throttling only the workloads that will minimize the objective function. Triage [17] keeps track of which performance band the system is operating in; it shares Façade's lack of selectivity, as a single QoS violation may bring the whole system down to a lower band (which is equivalent to throttling every workload). Sleds [7] can selectively throttle just the workloads supposedly responsible for the QoS violations, and has a decentralized architecture that scales better than Façade's. However, its policies for deciding which workload to throttle are hard-wired and will not adapt to changing conditions. Hippodrome [3] iteratively refines the data placement; each of its data migrations can take hours. It is a solution to longer-term problems than CHAMELEON, that is more appropriate for reacting in minutes to problems requiring immediate attention. Hippodrome can take a long time to converge (due to the high cost of migrating data) and can get stuck in local minima, for it relies on a variation of hill-climbing.

Scheduling-based approaches establish relative priorities between workloads and individual I/Os. Jin *et al.* [15] compared different scheduling algorithms for performance isolation and resource-usage efficiency; they found that scheduling is effective but cannot ensure tight bounds on the SLA constraints (essential for high-priority workloads). Stonehenge [12] uses a learning-based bandwidth allocation mechanism to map SLAs to virtual device shares dynamically; although it allows more general SLAs than CHAMELEON, it can only arbitrate accesses to the storage device, not to any other bottleneck component in the system. In general, scheduling approaches are designed to work well for the common case, not being effective in handling exception scenarios such as hardware failures.

Model-based approaches make decisions based on accurate models of the storage system. The main challenge is to build them, far from trivial in practical systems; system administrators very rarely have that level of information about the devices they use. Minerva [1] assumes that such models are given. CHAMELEON and Polus [25] (an extension of this vision) build those models on the fly, without supervision.

9 Conclusions

An ideal solution for resource arbitration in shared storage systems would adapt to changing workloads, client requirements and system conditions. It would also relieve system administrators from the burden of having to specify when to step in and take corrective action, and what actions to take—thus allowing them to concentrate on specifying the global objectives that maximize the storage utility's business benefit, and having the system take care of the details. No existing solution satisfies these criteria; prior approaches are either inflexible, or require administrators to supply up-front knowledge that is not available to them.

Our approach to identifying which client workloads should be throttled is based on constrained optimization. Constraints are derived from the running system, by monitoring its delivered performance as a function of the demands placed on it during normal operation. The objective function being optimized can be defined, and changed, by the administrator as a function of organizational goals. Given that the actions prescribed by our reasoning engine are only as good as the quality of the models used to compute them, CHAMELEON will switch to a conservative decision-making process if insufficient knowledge is available. CHAMELEON's approach to model building requires no prior knowledge about the quantitative characteristics of workloads and devices—and makes good decisions in realistic scenarios like those involving workloads with relative priorities. We replayed traces from production environments on a real storage system, and found that CHAMELEON makes very accurate decisions for the workloads examined. CHAMELEON always made the optimal throttling decisions, given the available knowledge. The times to react to and solve performance problems were in the 3-14 min. range, which is quite encouraging.

Areas for future work include component and

workload models that incorporate additional relevant parameters, more general (non-linear) optimizers to accommodate the resulting, more accurate problem formulations, and even some degree of workload prediction using techniques related to ARIMA [23].

Acknowledgments: We wish to thank Randy Katz, Jai Menon, Kaladhar Voruganti and Honesty Young for their inputs on the direction of this work and valuable comments on earlier versions of this paper. We also thank Lucy Cherkasova for her excellent shepherding. Finally, we thank the HP Labs Storage Systems Department for making their traces available to the general public.

References

- [1] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001.
- [2] E. Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-4, HP Laboratories, July 2001.
- [3] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *Proc. of Symposium on File and Storage Technologies (FAST)*, pages 175–188, January 2002.
- [4] BMC Software. *Patrol for Storage Networking*, 2004.
- [5] E. Borowsky, R. Golding, P. Jacobson, A. Merchant, L. Schreier, M. Spasojevic, and J. Wilkes. Capacity planning with phased workloads. In *Proceedings of the first international workshop on Software and performance*, pages 199–207. ACM Press, 1998.
- [6] Maria Calzarossa and Giuseppe Serazzi. Workload characterization: A survey. *Proc. IEEE*, 81(8):1136–1150, 1993.
- [7] D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. Lee. Performance virtualization for large-scale storage systems. In *Proceedings of the 22nd Symposium on Reliable Distributed Systems*, pages 109–118, October 2003.
- [8] Free Software Foundation, Inc., <http://www.gnu.org/software/glpk/glpk.html>. *GLPK (GNU Linear Programming Kit)*, 2003.
- [9] J.S. Glider, C. Fuente, and W.J. Scales. The software architecture of a san storage control system. *IBM Systems Journal*, 42(2):232–249, 2003.

- [10] Gartner Group. Total Cost of Storage Ownership—A User-oriented Approach. Research note, 2000.
- [11] Hewlett-Packard Laboratories, http://tesla.hpl.hp.com/public_software. Publicly-available software and traces, 2004.
- [12] Lan Huang, Gang Peng, and Tzi-cker Chiueh. Multi-dimensional storage virtualization. *SIGMETRICS Perform. Eval. Rev.*, 32(1):14–24, 2004.
- [13] IETF Policy Framework Working Group. IETF Policy Charter. <http://www.ietf.org/html.charters/policy-charter.html>.
- [14] Raj Jain. *The Art of Computer System Performance Analysis*. Wiley, 1991.
- [15] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. *SIGMETRICS Perform. Eval. Rev.*, 32(1):37–48, 2004.
- [16] T. Joachims. *Making large-scale SVM learning practical*. MIT Press, Cambridge, USA, 1998.
- [17] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. In *Proc. of the 12th. Int'l Workshop on Quality of Service*, June 2004.
- [18] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. Aqueduct: online data migration with performance guarantees. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 175–188, January 2002.
- [19] C. Lumb, A. Merchant, and G. A. Alvarez. Faç ade: virtual storage devices with performance guarantees. In *Proc. 2nd Conf. on File and Storage Technologies (FAST)*, pages 131–144, April 2003.
- [20] Storage Networking Industry Association, <http://www.snia.org>. *SMI Specification version 1.0*, 2003.
- [21] Storage Performance Council, <http://www.storageperformance.org>. *SPC I/O traces*, 2003.
- [22] David Sullivan. *Using probabilistic reasoning to automate software tuning*. PhD thesis, Harvard University, September 2003.
- [23] Nancy Tran and Daniel A. Reed. ARIMA time series modeling and forecasting for adaptive i/o prefetching. In *Proceedings of the 15th international conference on Supercomputing*, pages 473–485. ACM Press, 2001.
- [24] J. Turner. New directions in communications. *IEEE Communications*, 24(10):8–15, October 1986.
- [25] S. Uttamchandani, K. Voruganti, S. Srinivasan, J. Palmer, and D. Pease. Polus: Growing storage QoS management beyond a 4-year old kid. In *FAST04*, March 2004.
- [26] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proc. of the 9th Intl. Symp. on Modeling, Analysis and Simulation on Computer and Telecommunications Systems*, pages 183–192, August 2001.
- [27] D. Verma. Simplifying network administration using policy-based management. *IEEE Network Magazine*, 16(2), March 2002.
- [28] D. Verma and S. Calo. Goal Oriented Policy Determination. In *Proc. 1st Workshop on Algorithms and Architectures for Self-Managing Sys.*, pages 1–6. ACM, June 2003.
- [29] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with CART models. *SIGMETRICS Perform. Eval. Rev.*, 32(1):412–413, 2004.

A Transactional Flash File System for Microcontrollers

Eran Gal and Sivan Toledo*
School of Computer Science, Tel-Aviv University

Abstract

We present a transactional file system for flash memory devices. The file system is designed for embedded microcontrollers that use an on-chip or on-board NOR flash device as a persistent file store. The file system provides atomicity to arbitrary sequences of file system operations, including reads, writes, file creation and deletion, and so on. The file system supports multiple concurrent transactions. Thanks to a sophisticated data structure, the file system is efficient in terms of read/write-operation counts, flash-storage overhead, and RAM usage. In fact, the file system typically uses several hundreds bytes of RAM (often less than 200) and a bounded stack (or no stack), allowing it to be used on many 16-bit microcontrollers. Flash devices wear out; each block can only be erased a certain number of times. The file system manages the wear of blocks to avoid early wearing out of frequently-used blocks.

1 Introduction

We present TFFS, a transactional file system for flash memories. TFFS is designed for microcontrollers and small embedded systems. It uses extremely small amounts of RAM, performs small reads and writes quickly, supports general concurrent transactions and single atomic operations, and recovers from crashes reliably and quickly. TFFS also ensures long device life by evening out the wear of blocks of flash memory (flash memory blocks wear out after a certain number of erasures).

Flash memory is a type of electrically erasable programmable read-only memory (EEPROM). Flash memory is nonvolatile (retains its content without power), so it is used to store files and other persistent objects in workstations and servers (for the BIOS), in handheld computers and mobile phones, in digital cameras, and in portable music players.

The read/write/erase behavior of flash memory is radically different than that of other programmable memories, such as volatile RAM and magnetic disks. Perhaps more importantly, memory cells in a flash device (as well as in other types of EEPROMs) can be written to only a limited number of times, between 10,000 and 1,000,000, after which they wear out and become unreliable.

Flash memories come in three forms: on-chip memories in system-on-a-chip microcontrollers, standalone chips for board-level integration and removable memory devices (USB sticks, SmartMedia cards, CompactFlash cards, and so on). The file system that we present in this paper is designed for system-on-a-chip microcontrollers that include flash memories and for on-board standalone chips. The file system is particularly suited for devices with very little RAM (system-on-a-chip microcontrollers often include only 1-2 KB of RAM).

Flash memories also come in several flavors with respect to how reads and writes are performed (see, e.g. [1, 2]). The two main categories are NOR flash, which behaves much like a conventional EEPROM device, and NAND flash, which behaves like a block device. But even within each category there are many flavors, especially with respect to how writes are performed. Our file system is designed for NOR flash, and in particular, for devices that are *memory mapped and allow reprogramming*. That is, our file system assumes that four flash operations are available:

- Reading data through random-access memory instructions.
- Erasing a block of storage; in flash memories and EEPROM, erasing a block sets all the bits in the block to a logical '1'.
- Clearing one or more bits in a word (usually 1-4 bytes) consisting of all ones. This is called programming.
- Clearing one or more bits in a word with some zero bits. This is called reprogramming the word.

Virtually all NOR devices support the first three operations, and many support all four, but some do not support reprogramming.

Because flash memories, especially NOR flash, have very different performance characteristics than magnetic disks, file systems designed for disks are usually not appropriate for flash. Flash memories are random access devices that do not benefit at all from access patterns with temporal locality (rapid repeated access to the same location). NOR flash memories do not benefit from spatial locality in read and write accesses; some benefit from sequential access to blocks of a few bytes. Spatial locality is important in erasures—performance is best when much of the data in a block becomes obsolete roughly at the same time.

The only disk-oriented file systems that addresses at least some of these issues are log-structured file systems [3, 4], which indeed have been used on flash devices, often with flash-specific adaptations [5, 6, 7, 8]. But even log-structured file systems ignore some of the features of flash devices, such as the ability to quickly write a small chunk of data anywhere in the file system. As we shall demonstrate in this paper, by exploiting flash-specific features we obtain a much more efficient file system.

File systems designed for small embedded systems must contend with another challenge, extreme scarcity of resources, especially RAM. Many of the existing flash file systems need large amounts of RAM, usually at least tens of kilobytes, so they are not suitable for small microcontrollers (see, e.g., [9, 10]; the same appears to be true for TargetFFS, www.blunkmicro.com, and for smxFFS, www.smxinfo.com). Two recent flash file systems for TinyOS, an experimental operating system for sensor-network nodes, Matchbox [11, 12] and ELF [8], are designed for microcontrollers with up to 4 KB of RAM.

TFFS is a newly-designed file system for flash memories. It is efficient, ensures long device life, and supports general transactions. Section 3 details the design goals of TFFS, Section 4 explains its design, and Section 5 demonstrates, experimentally, that it does meet its quantitative goals. The design of TFFS is unique; in particular, it uses a new data structure, pruned versioned trees, that we developed specifically for flash file systems. Some of our goals are also novel, at least for embedded systems. Supporting general transactions is not a new idea in file systems [13, 14], but it is not common either; but general transactions were never supported in flash-specific file systems. Although transactions may seem like a luxury for small embedded systems, we believe that transaction support by the file system can simplify many applications and contribute to the reliability of embedded systems.

2 Flash Memories

This section provides background information on flash memories. For further information on flash memories, see, for example, [1] or [2]. We also cover the basic principles of flash-storage management, but this section does not survey flash file systems, data structure and algorithms; relevant citations are given throughout the paper. For an exhaustive coverage of these techniques, see our recent survey [15].

Flash memories are a type of electrically-erasable programmable read-only memory (EEPROM). EEPROM devices store information using modified MOSFET transistors with an additional floating gate. This gate is electrically isolated from the rest of the circuit, but it can

nonetheless be charged and discharged using a tunneling and/or a hot-electron-injection effect.

Traditional EEPROM devices support three types of operations. The device is memory mapped, so reading is performed using the processor's memory-access instructions, at normal memory-access times (tens of nanoseconds). Writing is performed using a special on-chip controller, not using the processor's memory-access instructions. This operation is usually called *programming*, and takes much longer than reading; usually a millisecond or more. Programming can only clear set bits in a word (flip bits from '1' to '0'), but not vice versa. Traditional EEPROM devices support *reprogramming*, where an already programmed word is programmed again. Reprogramming can only clear additional bits in a word. To set bits, the word must be *erased*, an operation that is also carried out by the on-chip controller. Erasures often take much longer than even programming, often half a second or more. The word size of traditional EEPROM, which controls the program and erase granularities, is usually one byte.

Flash memories, or more precisely, flash-erasable EEPROMs, were invented to circumvent the long erase times of traditional EEPROM, and to achieve denser layouts. Both goals are achieved by replacing the byte-erase feature by a block-erase feature, which operates at roughly the same speed, about 0.5–1.5 seconds. That is, flash memories also erase slowly, but each erasure erases more bits. Block sizes in flash memories can range from as low as 128 bytes to 64 KB. In this paper, we call these erasure blocks *erase units*.

Many flash devices, and in particular the devices for which TFFS is designed, differ from traditional EEPROMs only in the size of erase units. That is, they are memory mapped, they support fine-granularity programming, and they support reprogramming. Many of the flash devices that use the so-called NOR organization support all of these features, but some NOR devices do not support reprogramming. In general, devices that store more than one bit per transistor (MLC devices) rarely support reprogramming, while single-bit per transistor often do, but other factors may also affect reprogrammability.

Flash-device manufacturers offer a large variety of different devices with different features. Some devices support additional programming operations that program several words at a time; some devices have uniform-size erase blocks, but some have blocks of several sizes and/or multiple banks, each with a different block size; devices with NAND organization are essentially block devices—read, write, and erase operations are performed by a controller on fixed-length blocks. A single file-system design is unlikely to suit all of these devices. TFFS is designed for the most type of flash memories that are used in system-on-a-chip microcontrollers and

low-cost standalone flash chips: reprogrammable NOR devices.

Storage cells in EEPROM devices wear out. After a certain number of erase-program cycles, a cell can no longer reliably store information. The number of reliable erase-program cycles is random, but device manufacturers specify a guaranteed lower bound. Due to wear, the life of flash devices is greatly influenced by how it is managed by software: if the software evens out the wear (number of erasures) of different erase units, the device lasts longer until one of the units wears out.

Flash devices are used to store data objects. If the size of the data objects matches the size of erase units, then managing the device is fairly simple. A unit is allocated to an object when it is created. When the data object is modified, the modified version is first programmed into an erased unit, and then the previous copy is erased. A mapping structure must be maintained in RAM and/or flash to map application objects to erase units. This organization is used mostly in flash devices that simulate magnetic disks—such devices are often designed with 512-byte erase units that each stores a disk sector.

When data objects are smaller than erase units, a more sophisticated mechanism is required to reclaim space. When an object is modified, the new version is programmed into a not-yet-programmed area in some erase unit. Then the previous version is marked as obsolete. When the system runs out of space, it finds an erase unit with some obsolete objects, copies the still-valid objects in that unit to free space on other units, and erases the unit. The process of moving the valid data to other units, modifying the object mapping structures, and erasing a unit is called *reclamation*. The data objects stored on an erase unit can be of uniform size, or of variable size [16, 17].

3 Design Goals

We designed TFFS to meet the requirements of small embedded systems that need a general-purpose file system for NOR flash devices. Our design goals, roughly in order of importance, were

- Supporting the construction of highly-reliable embedded applications,
- Efficiency in terms of RAM usage, flash storage utilization, speed, and code size,
- High endurance.

Supporting general-purpose file-system semantics, such as the POSIX semantics, was not one of our goals. In particular, we added functionality that POSIX file systems do not support when this functionality served our goals,

and we do not support some POSIX features that would have reduced the efficiency of the file system.

The specification of the design goals of TFFS was driven by an industrial partner with significant experience in operating systems for small embedded systems. The industrial partner requested support for explicit concurrent transactions, requested that RAM usage be kept to a minimum, and designed with us the API of TFFS. We claim that this involvement of the industrial partner in the specification of TFFS demonstrates that our design goals serve genuine industrial needs and concerns.

Embedded applications must contend with sudden power loss. In any system consisting of both volatile and nonvolatile storage, loss of power may leave the file system itself in an inconsistent state, or the application's files in an inconsistent state from the application's own viewpoint. TFFS performs all file operations atomically, and the file system always recovers to a consistent state after a crash. Furthermore, TFFS's API supports explicit and concurrent transactions. Without transactions, all but the simplest applications would need to implement an application-specific recovery mechanism to ensure reliability. TFFS takes over that responsibility. The support for concurrent transactions allows multiple concurrent applications on the same system to utilize this recovery mechanism.

Some embedded systems ignore the power loss issue, and as a consequence are simply unreliable. For example, the ECI Telecom B-FOCuS 270/400PR router/ADSL modem presents to the user a dialog box that reads “[The save button] saves the current configuration to the flash memory. Do not turn off the power before the next page is displayed, or else *the unit will be damaged!*”. Similarly, the manual of the Olympus C-725 digital camera warns the user that loosing power while the flash-access lamp is blinking could destroy stored pictures.

Efficiency, as always, is a multifaceted issue. Many microcontrollers only have 1–2 KB of RAM, and in such systems, RAM is often the most constrained resource. As in any file system, maximizing the effective storage capacity is important; this usually entails minimizing internal fragmentation and the storage overhead of the file system's data structures. In many NOR flash devices, programming (writing) is much slower than reading, and erasing blocks is even slower. Erasure times of more than half a second are common. Therefore, speed is heavily influenced by the number of erasures, and also by overhead writes (writes other than the data writes indicated by the API). Finally, storage for code is also constrained in embedded systems, so embedded file systems need to fit into small footprints.

In TFFS, RAM usage is the most important efficiency metric. In particular, TFFS never buffers writes, in order

```

TID BeginTransaction();
int CommitTransaction(tid);
int AbortTransaction(tid);
FD Open(FD parent, uint16 name, tid);
FD Open(char* long_name, tid);
FD CreateFile(type, name, long_name[],
             properties, FD parent_dir, tid);
int ReadBinary(file, buffer, length,
              offset, tid);
int WriteBinary(...);
int ReadRecord(file, buffer, length,
              record_number, tid);
int UpdateRecord(...);
int AddRecord(file, buff, length, tid);
int CloseFile(file, tid);
int DeleteFile(file);

```

Figure 1: A slightly simplified version of the API of TFFS. The types TID and FD stand for transaction identifier and file descriptor (handle), respectively. We do not show the type of arguments when the type is clear from the name (e.g., `char*` for `buffer`, `FD` for `file`, and so on). We also do not show a few utility functions, such as a function to retrieve a file's properties.

not to use buffer space. The RAM usage of TFFS is independent of the number of resources in use, such as open files. This design decision trades off speed for RAM. For small embedded systems, this is usually the right choice. For example, recently-designed sensor-network nodes have only 0.5–4 KB of RAM, so file systems designed for them must contend, like TFFS, with severe RAM constraints [8, 11, 12]. We do not believe that RAM-limited systems will disappear anytime soon, due to power issues and to mass-production costs; even tiny 8-bit microcontrollers are still widely used.

The API of the file system is nonstandard. The API, presented in a slightly simplified form in Figure 1, is designed to meet two main goals: support for transactions, and efficiency. The efficiency concerns are addressed by API features such as support for integer file names and for variable-length record files. In many embedded applications, file names are never presented to the user in a string form; some systems do not have a textual user interface at all, and some do, but present the files as nameless objects, such as appointments, faxes, or messages. Variable-length record files allow applications to efficiently change the length of a portion of a file without rewriting the entire file.

We deliberately excluded some common file-system features that we felt were not essential for embedded system, and which would have complicated the design or would have made the file system less efficient. The most important among these are directory traversals, file

truncation, and changing the attributes of a file (e.g., its name). TFFS does not support these features. Directory traversals are helpful for human users; embedded file systems are used by embedded applications, so the file names are embedded in the applications, and the applications know which files exist.

One consequence of the exclusion of these features is that the file system cannot be easily integrated into some operating systems, such as Linux and Windows CE. Even though these operating systems are increasingly used in small embedded systems, such as residential gateways and PDAs, we felt that the penalty in efficiency and code size to support general file-system semantics would be unacceptable for smaller devices.

On the other hand, the support for transactions does allow applications to reliably support features such as long file names. An application that needs long names for files can keep a long-names file with a record per file. This file would maintain the association between the integer name of a file and its long file name, and by creating the file and adding a record to the naming file in a transaction, this application data structure would always remain consistent.

The endurance issue is unique to flash file systems. Since each block can only be erased a limited number of times, uneven wear of the blocks leads to early loss of storage capacity (in systems that can detect and not use worn-out blocks), or to an untimely death of the entire system (if the system cannot function with some bad blocks).

We will show below that TFFS does meet these objectives. We will show that the support for transactions is correct, and we will show experimentally that TFFS is efficient and avoids early wear. Because we developed the API of TFFS with an industrial partner, we believe that the API is appropriate.

4 The Design of the File System

4.1 Logical Pointers and the Structure of Erase Units

The memory space of flash devices is partitioned into *erase units*, which are the smallest blocks of memory that can be erased. TFFS assumes that all erase units have the same size. (In some flash devices, especially devices that are intended to serve as boot devices, some erase units are smaller than others; in some cases the irregularity can be hidden from TFFS by the flash device driver, which can cluster several small units into a single standard-size one, or TFFS can ignore the irregular units.) TFFS reserves one unit for the log, which allows it to perform transactions atomically. The structure of this erase unit is simple: it is treated as an array of fixed-size records, which TFFS

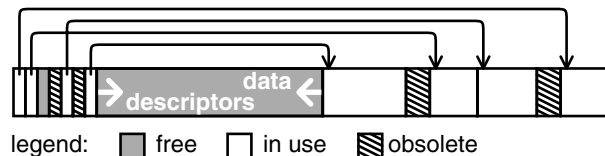


Figure 2: Partitioning an erase unit into variable-length sectors.

always fills in order. The other erase units are all used by TFFS's memory allocator, which uses them to allocate variable-sized blocks of memory that we call *sectors*.

The on-flash data structure that the memory allocator uses is designed to achieve one primary goal. Suppose that an erase unit that still contains valid data is selected for erasure, perhaps because it contains the largest amount of obsolete data. The valid data must be copied to another unit prior to the unit's erasure. If there are pointers to the physical location of the valid data, these pointers must be updated to reflect the new location of the data. Pointer modification poses two problems. First, the pointers to be modified must be found. Second, if these pointers are themselves stored on the flash device, they cannot be modified in place, so the sectors that contain them must be rewritten elsewhere, and pointers to them must now be modified as well. The memory allocator's data structure is designed so that pointer modification is never needed.

TFFS avoids pointer modification by using *logical pointers* to sectors rather than physical pointers; pointers to addresses within sectors are not stored at all. A logical pointer is an unsigned integer (usually a 16-bit integer) consisting of two bit fields: a logical erase unit number and a sector number. When valid data in a unit is moved to another unit prior to erasure, the new unit receives the logical number of the unit to be erased, and each valid sector retains its sector number in the new unit.

A table, indexed by logical erase-unit number, stores logical-to-physical erase-unit mapping. In our implementation the table is stored in RAM, but it can also be stored in a sector on the flash device itself, to save RAM.

Erase units that contain sectors (rather than the log) are divided into four parts, as shown in Figure 2. The top of the unit (lowest addresses) stores a small header, which is immediately followed by an array of sector descriptors. The bottom of the unit contains sectors, which are stored contiguously. The area between the last sector descriptor and the last sector is free. The sector area grows upwards, and the array of sector descriptors grows downwards, but they never collide. Area is not reserved for sectors or descriptors; when sectors are small more area is used by the descriptors than when sectors are large. A sector descriptor contains the erase-unit offset of first address in the sector, as well as a valid bit and an obsolete

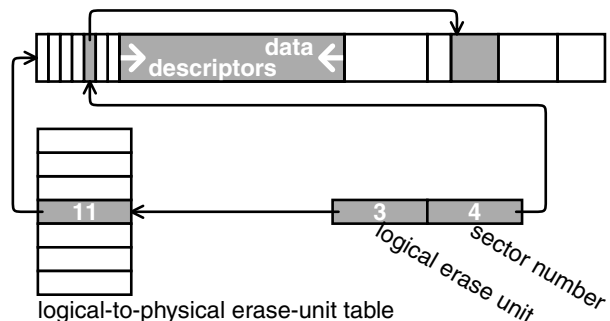


Figure 3: Translating a logical pointer into a physical address. The logical pointer consists of a logical erase unit (3 in the figure) and a sector number (4 in the figure). The logical-to-physical erase-unit table maps logical unit 3 to physical unit 11, which is the erase unit shown in the figure. The pointer stored in descriptor number 4 in erase unit 11 points to the address represented by the logical pointer.

bit. Clearing the valid bit indicates that the offset field has been written completely (this ensures that sector descriptors are created atomically). Clearing the obsolete bit indicates that the sector that the descriptor refers to is now obsolete.

A logical pointer is translated to a physical pointer as shown in Figure 3. The logical erase-unit number within the logical pointer is used as an index into the erase-unit table. This provides the physical unit that contains the sector. Then the sector number within the logical pointer is used to index into the sector-descriptors array on that physical unit. This returns a sector descriptor. The offset in that descriptor is added to the address of the physical erase unit to yield the physical address of the sector.

Before an erase unit is erased, the valid sectors on it are copied to another unit. Logical pointers to these sectors remain valid only if the sectors retain their sector number on the new unit. For example, sector number 6, which is referred to by the seventh sector descriptor in the sector-descriptors array, must be referred to by the seventh sector descriptor in the new unit. The offset of the sector within the erase unit can change when it is copied to a new unit, but the sector descriptor must retain its position. Because all the valid sectors in the unit to be erased must be copied to the same unit, and since specific sector numbers must be available in that unit, TFFS always copies sectors to a fresh unit that is completely empty prior to erasure of another unit. Also, TFFS always compacts the sectors that it copies in order to create a large contiguous free area in the new unit.

TFFS allocates a new sector in two steps. First, it finds an erase unit with a large-enough free area to accommodate the size of the new sector. Our implementation uses

a unit-selection policy that combines a limited-search best-fit approach with a classification of sectors into frequently and infrequently changed ones. The policy attempts to cluster infrequently-modified sectors together in order to improve the efficiency of erase-unit reclamation (the fraction of the obsolete data on the unit just prior to erasure). Next, TFFS finds on the selected unit an empty sector descriptor to refer to the sector. Empty descriptors are represented by a bit pattern of all 1's, the erased state of the flash. If all the descriptors are used, TFFS allocates a new descriptor at the bottom of the descriptors array. (TFFS knows whether all the descriptors are used in a unit; if they are, the best-fit search ensures that the selected unit has space for both the new sector and for a new descriptor).

The size of the sector-descriptors array of a unit is not represented explicitly. When a unit is selected for erasure, TFFS determines the size using a linear downwards traversal of the array, while maintaining the minimal sector offset that a descriptor refers too. When the traversal reaches that location, the traversal is terminated. The size of sectors is not represented explicitly, either, but it is needed in order to copy valid sectors to the new unit during reclamations. The same downwards traversal is also used by TFFS to determine the size of each sector. The traversal algorithm exploits the following invariant properties of the erase-unit structure. Sectors and their descriptors belong to two categories: *reclaimed sectors*, which are copied into the unit during the reclamation of another unit, and *new sectors*, allocated later. *Within each category, sectors with consecutive descriptors are adjacent to each other.* That is, if descriptors i and $j > i$ are both reclaimed or both new, and if descriptors $i + 1, \dots, j - 1$ all belong to the other category, then sector i immediately precedes sector j . This important invariant holds because (1) we copy reclaimed sectors from lower-numbered descriptors to higher numbered ones, (2) we always allocated the lowest-numbered free descriptor in a unit for a new sector, and (3) we allocate the sectors themselves from the top down (from right to left in Figure 2). The algorithm keeps track of two descriptor indices, ℓ_r , the reclaimed descriptor, and ℓ_n , the last new descriptor. When the algorithm examines a new descriptor i , it first determines whether it is free (all 1's), new or reclaimed. If it is free, the algorithm proceeds to the next descriptor. Otherwise, if the sector lies to the right of the last-reclaimed mark stored in the unit's header, it is reclaimed, otherwise new. Suppose that i is new; sector i starts at the address given by its header, and it ends at the last address before ℓ_n , or the end of the unit if i is the first new sector encountered so far. The case of reclaimed sectors is completely symmetric. Note that the traversal processes both valid and obsolete sectors.

As mentioned above, each erase unit starts with a header. The header indicates whether the unit is free, used for the log, or for storing sectors. The header contains the logical unit that the physical unit represents (this field is not used in the log unit), and an erase counter. The header also stores the highest (leftmost) sector offset of sectors copied as part of another unit's reclamation process; this field allows us to determine the size of sectors efficiently. Finally, the header indicates whether the unit is used for storing frequently- or infrequently-modified data; this helps cluster related data to improve the efficiency of reclamation. In a file system that uses n physical units of m bytes each, and with an erase counter bounded by g , the size of the erase-unit header in bits is $3 + \lceil \log_2 n \rceil + \lceil \log_2 m \rceil + \lceil \log_2 g \rceil$. Flash devices are typically guaranteed for up to one million erasures per unit (and often less, around 100,000), so an erase counter of 24 bits allows accurate counting even if the actual endurance is 16 million erasures. This implies that the size of the header is roughly $27 + \log_2(nm)$, which is approximately 46 bits for a 512 KB device and 56 bits for a 512 MB device.

The erase-unit headers represent an on-flash storage overhead that is proportional to the number of units. The size of the logical-to-physical erase-unit mapping table is also proportional to the number of units. Therefore, a large number of units causes a large storage overhead. In devices with small erase units, it may be advantageous to use a flash device driver that aggregates several physical units into larger ones, so that TFFS uses a smaller number of larger units.

4.2 Efficient Pruned Versioned Search Trees

TFFS uses a novel data structure that we call *efficient versioned search trees* to support efficient atomic file-system operations. This data structure is a derivative of persistent search trees [18, 19], but it is specifically tailored to the needs of file systems. In TFFS, each node of a tree is stored in a variable-sized sector.

Trees are widely-used in file systems. For example, Unix file systems use a tree of indirect blocks, whose root is the inode, to represent files, and many file systems use search trees to represent directories. When the file system changes from one state to another, a tree may need to change. One way to implement atomic operations is to use a *versioned tree*. Abstractly, the versioned tree is a sequence of versions of the tree. Queries specify the version that they need to search. Operations that modify the tree always operate on the most recent version. When a sequence of modifications is complete, an explicit commit operation freezes the most recent version, which becomes read-only, and creates a new read-write

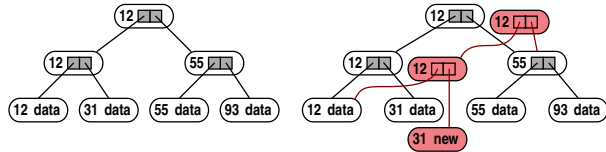


Figure 4: Path copying. Replacing the data associated with the leaf whose key is 31 in a binary tree (left) creates a new leaf (right). The leaf replacement propagates up to the root. The new root represents the new version of the tree. Data structure items that are created as a result of the leaf modification are shown in red.

version.

When versioned trees are used to implement file systems, usually only the read-write version (the last one) and the read-only version that precedes it are accessed. The read-write version represents the state of the tree while processing a transaction, and the read-only version represents the most-recently committed version. The read-only versions satisfy all the data-structure invariants; the read-write version may not. Because old read-only versions are not used, they can be pruned from the tree, thereby saving space. We call versioned trees that restrict read access to the most recently-committed version *pruned versioned trees*.

The simplest technique to implement versioned trees is called *path copying* [18, 19], illustrated in Figure 4. When a tree node is modified, the modified version cannot overwrite the existing node, because the existing node participates in the last committed version. Instead, it is written elsewhere in memory. This requires a modification in the parent as well, to point to the new node, so a new copy of the parent is created as well. This always continues until the root. If a node is modified twice or more before the new version is committed, it can be modified in place, or a new copy can be created in each modification. If the new node is stored in RAM, it is usually modified in place, but when it is stored on a difficult to modify memory, such as flash, a new copy is created. The log-structured file system [3, 4], for example, represents each file as tree whose root is an inode, and uses this algorithm to modify files atomically. WAFL [20], a file system that supports snapshots, represents the entire file-system as a single tree, which is modified in discrete write episodes; WAFL maintains several read-only versions of the file-system tree to provide users with access to historical states of the file system.

A technique called *node copying* can often prevent the copying of the path from a node to the root when the node is modified, as shown in Figure 5. This technique relies on *spare pointers* in tree nodes, and on nodes that can be physically modified in place. To implement node copy-

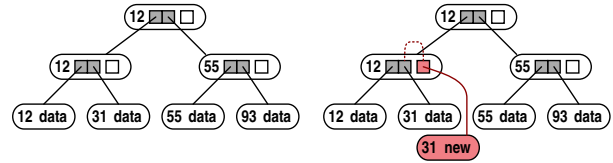


Figure 5: Node copying. Internal tree nodes contain a spare pointer, initially unused (white). Replacing a leaf creates a new leaf and sets the spare pointer in its parent. The spare pointer points to the new leaf, and it also indicates the child pointer that it replaced (dotted line).

ing, nodes are allocated with one or more spare pointers, which are initially empty. When a child pointer in a node needs to be updated, the system determines whether the node still contains an empty spare pointer. If it does, the spare pointer is modified instead. The modified spare pointer points to the new child, and it contains an indication of which original pointer it replaces.

Each spare pointer also includes a commit bit, to indicate whether it has been created in the current read-write version or in a previous version. If the commit bit is set, then tree accesses in both the read-only version and the read-write version should traverse the spare pointer, not the original pointer that it replaces. If the commit bit is not yet set, then tree access in the read-write version should traverse the spare pointer but tree access in the read-only version should traverse the original pointer. Spare pointers also have an abort bit; if set, then the spare pointer is simply ignored.

In B-trees, in which nodes have a variable number of child pointers, the spare pointer can also be used to add a new child pointer. This serves two purposes. First, it allows us to allocate variable-size nodes, containing only enough child pointers for the number of children the node has at creation time. Second, it allows us to store original child pointers without commit bits.

In principle, using a number of spare pointers can further reduce the number of node creations, at the expense of more storage overhead. However, even with only one spare, it can be shown that the amortized cost of a single tree update/insert/delete operation is constant. Therefore, our file system always uses only one spare per node.

4.3 Tree Traversals with a Bounded Stack

Small embedded systems often have very limited stack space. Some systems do not use a stack at all: static compiler analysis maps automatic variables to static RAM locations. To support systems with a bounded or no stack, TFFS never uses explicit recursion. We do use recursive algorithms, but the recursion uses a statically-allocated

stack and its depth is configured at compile time. We omit further details.

4.4 Mapping Files and File Names

TFFS uses pruned versioned trees for mapping files and file names. Most of the trees represent files and directories, one tree per file/directory. These trees are versioned.

In record files, each record is stored in a separate sector, and the file's tree maps record numbers to the logical addresses of sectors. In binary files, extents of contiguous data are stored on individual sectors, and the file's tree maps file offsets to sectors. The extents of binary files are created when data is appended to the file. Currently, TFFS does not change this initial partitioning.

TFFS supports two naming mechanisms for the open system call. One mechanism is a hierarchical name space of directories and files, as in most file systems. However, in TFFS directory entries are short unsigned integers, not strings, in order to avoid string comparisons in directory searches. The second mechanism is a flat namespace consisting of unique strings. A file or directory can be part of one name space or both. In the future, we may merge these two mechanisms, as explained below. Currently, however, the hierarchical name space does not allow long names.

Therefore, directory trees are indexed by the short integer entry names. The leaves of directory trees are the metadata records of the files. The metadata record contains the internal file identifier (GUID) of the directory entry, as well as the file type (record/binary/directory), the optional long name, permissions, and so on. In TFFS, the metadata is immutable.

TFFS assumes that long names are globally unique. We use a hash function to map these string names to 16-bit integers, which are perhaps not unique. TFFS maps a directory name to its metadata record using a search tree indexed by hash values. The leaves of this tree are either the metadata records themselves (if a hash value maps into a single directory name), or arrays of logical pointers to metadata records (if the names of several directories map into the same hash value).

In the future, we may replace the indexing in directory trees from the explicit integer file names to hash values of long file names.

A second search tree maps GUIDs to file/directory trees. This tree is indexed by GUID and its leaves are logical pointers to the roots of file/directory trees.

The open system call comes in two versions: one returns a GUID given a directory name, and the other returns a GUID given a directory GUID and a 16-bit file identifier within that directory. The first computes the hash value of the given name and uses it to search the directory-names tree. When it reaches a leaf, it verifies

the directory name if the leaf is the metadata of a directory, or searches for a metadata record with the appropriate name if the leaf is an array of pointers to metadata records. The second variant of the system call searches the GUID tree for the given GUID of the directory. The leaf that this search returns is a logical pointer to the root of the directory tree. The system call then searches this directory tree for the file with the given identifier; the leaf that is found is a logical pointer to the metadata record of the sought-after file. That metadata record contains the GUID of the file.

In file-access system calls, the file is specified by a GUID. These system calls find the root of the file's tree using the GUID tree.

4.5 Transactions on Pruned Versioned Trees

The main data structures of TFFS are pruned versioned trees. We now explain how transactions interact with these trees. By transactions we mean not only explicit user-level transactions, but also implicit transactions that perform a single file-system modification atomically.

Each transaction receives a transaction identifier (TID). These identifiers are integers that are allocated in order when the transaction starts, so they also represent discrete time stamps. A transaction with a log TID time stamp started before a transaction with a higher TID. The file system can commit transactions out of order, but the linearization order of the transactions always corresponds to their TID: a transaction with TID t can observe all the side effects of committed transactions $t - 1$ and lower, and cannot observe any of the side effects of transactions $t + 1$ and higher.

When a transaction modifies a tree, it creates a new version of the tree. That version remains active, in read-write mode, until the transaction either commits or aborts. If the transaction commits, all the spare pointers that it created are marked as committed. In addition, if the transaction created a new root for the file, the new root becomes active (the pointer to the tree's root, somewhere else in the file system, is updated). If the transaction aborts, all the spare pointers that the transaction created are marked as aborted by a special bit. Aborted spare pointers are not valid and are never dereferenced.

Therefore, a tree can be in one of two states: either with uncommitted and unexpired spare pointers (or an uncommitted root), or with none. A tree in the first state is being modified by a transaction that is not yet committed or aborted. Suppose that a tree is being modified by transaction t , and that the last committed transaction that modified it is s . The read-only version of the tree, consisting of all the original child pointers and all the committed spare pointers, represents the state of the tree

in discrete times r for $s \leq r < t$. We do not know what the state of the tree was at times smaller than s : perhaps some of the committed spares represent changes made earlier, but we cannot determine when, so we do not know whether to follow them or not. Also, some of the nodes that existed at times before s may cease to exist at time s . The read-write version of the tree represents the state of the tree at time t , but only if transaction t will commit. If transaction t will abort, then the state of the tree at time t is the same state as at time s . If transaction t will commit, we still do not know what the state of the tree will be at time $t + 1$, because transaction t may continue to modify it. Hence, the file system allows transactions with TID r , for $s < r < t$, access to the read-only version of the tree, and to transaction t access to the read-write version. All other access attempts cause the accessing transaction to abort. In principle, instead of aborting transactions later than t , TFFS could block them, but we assume that the operating system's scheduler cannot block a request.

If a tree is in the second state, with only committed or aborted spares, we must keep track not only of its last modification time s , but also of the latest transaction $u \geq s$ that read it. The file system admits read requests from any transaction r for $r > s$, and write requests from a transaction $t \geq u$. As before, read requests from a transaction $r < s$ causes r to abort. A write request from a transaction $t < u$ causes t to abort, because it might affect the state of the tree that u already observed.

To enforce these access rules, we associate three TIDs with each versioned tree: the last committed modification TID, the last read-access TID, and the TID that currently modifies the tree, if any. These TIDs are kept in a search tree, indexed by the internal identifier of the versioned tree. The file system never accesses the read-only version of the TID tree. Therefore, although it is implemented as a pruned versioned tree, the file system treats it as a normal mutable search tree. The next section presents an optimization that allows the file system not to store the TIDs associated with a tree.

4.6 Using Bounded Transaction Identifiers

To allow TFFS to represent TIDs in a bounded number of bits, and also to save RAM, the file system represents TIDs modulo a small number. In essence, this allows the file system to store information only on transactions in a small window of time. Older transactions than this window permits must be either committed or aborted.

The TID allocator consists of three simple data structures that are kept in RAM: next TID to allocate, the oldest TID in the TID tree, and a bit vector with one bit per TID within the current TID window. The bit vector stores, for each TID that might be represented in the system,

whether it is still active or whether it has already been aborted or committed. When a new TID needs to be allocated, the allocator first determines whether the next TID represents an active transaction. If it does, the allocation simply fails. No new transactions can be started until the oldest one in the system either commits or aborts. If the next TID is not active and not in the TID tree, it is allocated and the next-TID variable is incremented (modulo the window size). If the next TID is not active but it is in the TID tree, the TID tree is first cleaned, and then the TID is allocated.

Before cleaning the TID tree, the file system determines how many TIDs can be cleaned. Cleaning is expensive, so the file system cleans on demand, and when it cleans, it cleans as many TIDs as possible. The number of TIDs that can be cleaned is the number of consecutive inactive TIDs in the oldest part of the TID window. After determining this number, the file system traverses the entire TID tree and invalidates the appropriate TIDs. An invalid TID in the tree represents a time before the current window; transactions that old can never abort a transaction, so the exact TID is irrelevant. We cannot search for TIDs to be invalidated because the TID tree is indexed by the identifiers of the trees, not by TID.

The file system can often avoid cleaning the TID tree. Whenever no transaction is active, the file system deletes the entire TID tree. Therefore, if long chains of concurrent transactions are rare, tree cleanup is rare or not performed at all. The cost of TID cleanups can also be reduced by using a large TID window size, at the expense of slight storage inefficiency.

4.7 Atomic Non-transactional Operations

To improve performance and to avoid running out of TIDs, the file system supports non-transactional operations. Most requests to the file system specify a TID as an argument. If no TID is passed to a system call (the TID argument is 0), the requested operation is performed atomically, but without any serializability guarantees. That is, the operation will either success completely, or will fail completely, but it may break the serializability of concurrent transactions.

The file system allows an atomic operation to modify a file or directory only if no outstanding transaction has already modified the file's tree. But this still does not guarantee serializability. Consider a sequence of operations in which a transaction reads a file, which is subsequently modified by an atomic operation, and then read or modified again by the transaction. It is not possible to serialize the atomic operation and the transaction.

Therefore, it is best to use atomic operations only on files/directories that do not participate in outstanding transactions. An easy way to ensure this is to access a

particular file either only in transactions or only in atomic operations.

Atomic operations are more efficient than single-operation transactions in two ways. First, during an atomic operation the TID tree is read, to ensure that the file is not being modified by an outstanding transaction, but the TID tree is not modified. Second, a large number of small transactions can cause the file system to run out of TIDs, if an old transaction remains outstanding; atomic operations avoid this possibility, because they do not use TIDs at all.

4.8 The Log

TFFS uses a log to implement transactions, atomic operations, and atomic maintenance operations. As explained above, the log is stored on a single erase unit as an array of fixed-size records that grows downwards. The erase unit containing the log is marked as such in its header. Each log entry contains up to four items: a valid/obsolete bit, an entry-type identifier, a transaction identifier, and a logical pointer. The first two are present in each log entry; the last two remain unused in some entry types.

TFFS uses the following log-record types:

- **New Sector and New Tree Node.** These record types allow the system to undo a sector allocation by marking the pointed-to sector as obsolete. The New-Sector record is ignored when a transaction is committed, but a The New-Tree-Node record causes the file system to mark the spare pointer in the node, if used, as committed. This ensures that a node that is created in a transaction and modified in the same transaction is marked correctly.
- **Obsolete Sector.** Sectors are marked as obsolete only when the transaction that obsoleted them is committed. This node is ignored at abort time, and clears the obsolete bit of the sector at commit time.
- **Modified Spare Pointer.** Points to a node whose spare pointer has been set. Clears the spare's commit bit at commit time or its abort bit at abort time.
- **New File.** Points to the root of a file tree that was created in a transaction. At commit time, this record causes the file to be added to the GUID tree and to the containing directory. Ignored at abort time.
- **File Root.** Points to the root of a file tree, if the transaction created a new root. At commit time, the record is used to modify the file's entry in the GUID tree. Ignored at abort time.
- **Commit Marker.** Ensures that the transaction is redone at boot time.

- **Erase Marker.** Signifies that an erase unit is about to be erased. The record contains a physical erase-unit number and an erase count, but does not contain a sector pointer or a TID. This record typically uses two fixed-size record slots. If the log contains a non-obsolete erase-marker record at boot time, the physical unit is erased again; this completes an interrupted erasure.
- **GUID-Tree Pointer, TID-Tree Pointer, and Directory-Hash-Tree Pointer.** These records are written to the log when the root of one of these trees moves, to allow the file system to find them at boot time.

File trees are modified during transactions and so does the TID tree. The GUID and directory-hash trees, and directory trees, however, are only modified during commits. We cannot modify them during transactions because our versioned trees only support one outstanding transaction. Delaying the tree modification to commit time allows multiple outstanding transactions to modify a single directory, and allows multiple transactions to create files and directories (these operations affect the GUID and the directory-hash trees). TFFS does not allow file and directory deletions to be part of explicit transactions because that would have complicated the file/directory creation system calls.

The delayed operations are logged but not actually performed on the trees. After the commit system call is invoked, but before the commit marker is written to the log, the delayed operations are performed.

When a transaction accesses a tree whose modification by the same transaction may have been delayed, the tree access must scan the log to determine the actual state of the tree, from the viewpoint of that transaction. Many of these log scans are performed in order to find the roots of files that were created by the transaction or whose root was moved by the transaction. To locate the roots of these file trees more quickly, the file system keeps a cache of file roots that were modified by the transaction. If a file that is accessed is marked in the TID tree as being modified by the transaction, the access routine first checks this cache. If the cache contains a pointer to the file's root, the search in the log is avoided; otherwise, the log is scanned for a non-obsolete file-root record.

5 Implementation and Performance

This section describes the implementation of the file system and its performance. The performance evaluation is based on detailed simulations that we performed using several simulated workloads. The simulations measure the performance of the file system, its storage overheads, its endurance, and the cost of leveling the device's wear.

5.1 Experimental Setup

This section describes the experimental setup that we used for the simulations.

Devices. We performed the experiments using simulations of two real-world flash devices. The first is an 8 MB stand-alone flash-memory chip, the M29DW640D from STMicroelectronics. This device consists of 126 erase units of 64 KB each (and several smaller ones, which our file system does not use), read access times of about 90 ns, program times of about 10 us, and block-erase times of about 0.8 seconds.

The second device is a 16-bit microcontroller with on-chip flash memory, the ST10F280, also from STMicroelectronics. This chip comes with two banks of RAM, one containing 2 KB and the other 16 KB, and 512 KB of flash memory. The flash memory contains 7 erase units of 64 KB each (again, with several smaller units that we do not use). The flash access times are 50 ns for reads, 16 us for writes, and 1.5 seconds erases. The small number of erase units in this chip hurts TFFS's performance; to measure the effect, we also ran simulations using this device but with smaller erase units ranging from 2 to 32 KB.

Both devices are guaranteed for 100,000 erase cycles per erase unit.

File-System Configuration. We configured the non-hardware-related parameters of the file system as follows. The file system is configured to support up to 32 concurrent transactions, B-tree nodes have either 2–4 children or 7–14 children, 10 simulated recursive-call levels, and a RAM cache of 3 file roots. This configuration requires 466 bytes of RAM for the 8 MB flash and 109 bytes for the 0.5 MB flash.

Workloads. We used 3 workloads typical of flash-containing embedded systems to evaluate the file system. The first workload simulates a fax machine. This workload is typical not only of fax machines, but of other devices that store fairly large files, such as answering machines, dictating devices, music players, and so on. The workload also exercises the transactions capability of the file system. This workload contains:

- A parameter file with 30 variable length records, ranging from 4 to 32 bytes (representing the fax's configuration). This file is created, filled, and is never touched again.
- A phonebook file with 50 fixed-size records, 32 bytes each. This file is also created and filled but never accessed again.

- Two history files consisting of 200 cyclic fixed-size records each. They record the last 200 faxes sent and 200 last faxes received. They are changed whenever a fax page is sent or received.
- Each arriving fax consists of 4 pages, 51,300 bytes each. Each page is stored in a separate file and the pages of each fax are kept in a separate directory that is created when the fax arrives. The arrival of a fax triggers a transaction that creates a new record in the history file and creates a new directory for the file. The arrival of every new fax page adds changes the fax's record in the history file and creates a new file. Data is written to fax-page files in blocks of 1024 bytes.
- The simulation does not include sending faxes.

We also ran experiments without transactions under this workload, in order to assess the extra cost of transactions. We did not detect any significant differences when no transactions were used, so we do not present these results in the paper.

The second workload simulates a cellular phone. This simulation represents workloads that mostly store small files or small records, such as beepers, text-messaging devices, and so on. This workload consists of the following files and activities:

- Three 20-record cyclic files with 15-byte records, one for the last dialed numbers, one for received calls, and one for sent calls.
- Two SMS files, one for incoming messages and one for outgoing messages. Each variable-length record in these files stores one message.
- An appointments file, consisting of variable-length records.
- An address book file, consisting of variable-length records.
- The simulation starts by adding to the phone 150 appointments and 50 address book entries.
- During the simulation, the phone receives and sends 3 SMS messages per day (3 in each direction), receives 10 and dials 10 calls, and misses 5 calls, adds 5 new appointments and deletes the oldest 5 appointments.

The third workload simulates an event recorder, such as a security or automotive "black box", a disconnected remote sensor, and so on. The simulation represents workloads with a few event-log files, some of which record frequent events and some of which record rare events (or perhaps just the extreme events from a high-frequency event stream). This simulation consists of three files:

- One file records every event. This is a cyclic file with 32-byte records.

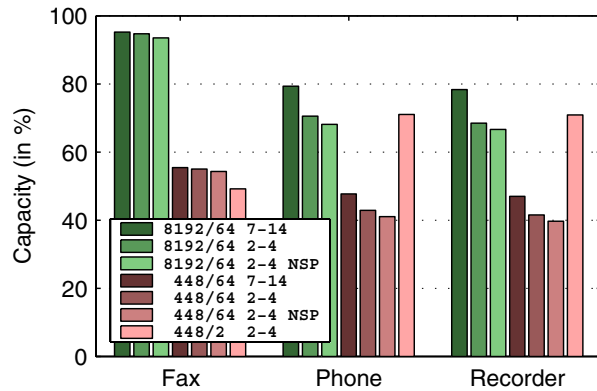


Figure 6: The capacity of TFFS. For each workload, the graph shows 7 bars: 3 for an 8 MB flash with 64 KB erase units (denoted by 8192/64) and 4 bars for a 448 KB flash with either 64 or 2 KB erase units (448/64 and 448/2). Two bars are for file systems whose B-trees have 7–14 children, and the rest for B-trees with 2–4 children. The scenarios denoted NSP describe a file system which does not use spare pointers.

- The other file records one event per 10 full cycles through the other files. This file too is cyclic with 32-byte records.
- A configuration file with 30 variable-size records ranging from 4 to 32 bytes. These files are filled when the simulation starts and never accessed again.

5.2 Capacity Experiments

Figure 6 presents the results of experiments intended to measure the storage overhead of TFFS. In these simulations, we initialize the file system and then add data until the file system runs out of storage. In the fax workload, we add 4-page faxes to the file system until it fills. In the phone workload, we do not erase SMS messages. In the event-recording simulation, we replace the cyclic files by non-cyclic files.

The graph shows the amount of user data written to the file system before it ran out of flash storage, as a percentage of the total capacity of the device. For example, if 129,432 bytes of data were written to a flash file system that uses a 266,144 bytes flash, the capacity is 49%.

The groups of bars in the graph represent different device and file-system configurations: an 8 MB device with 64 KB erase units, a 448/64 KB device, and a 448/2 KB device; file systems with 2–4 children per tree node and file systems with 7–14 children; file systems with spare pointers and file systems with no spare pointers.

Clearly, storing large extents, as in the fax workload, reduces storage overheads compared to storing

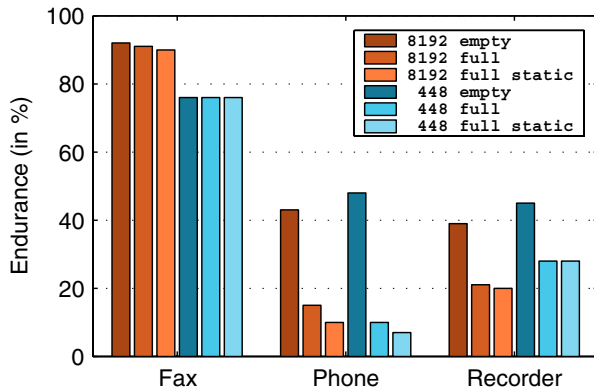


Figure 7: Endurance under different contents scenarios. For each flash size, the graph shows the endurance of a file system that is always almost empty, for a file system that is always almost full and half its data is static, and for a full file system with almost only static data.

small records or extents. Wider tree nodes reduce overheads when the leaves are small. The performance- and endurance-oriented experiments that we present later, however, indicate that wider nodes degrade performance and endurance. A small number of erase units leads to high overheads. Small erase units reduce overheads except in the fax workload, in which the 1 KB extents fragment the 2 KB erase units.

5.3 Endurance Experiments

The next set of experiments measures both endurance, which we present here, and performance, which we present in the next section. All of these experiments run until one of the erase units reaches an erasure count of 100,000; at that point, we consider the device worn out. We measure endurance by the amount of user data written to the file system as a percentage of the theoretical endurance limit of the device. For example, a value of 68 means that the file system was able to write 68% of the data that can be written on the device if wear is completely even and if only user data is written to the device.

We performed two groups of experiment. The first assesses the impact of file-system fullness and data life spans on TFFS's behavior. In particular, we wanted to understand how TFFS copes with a file system that is almost full and with a file system that contains a significant amount of static data. This group consists of three scenarios: one scenario in which the file system remains mostly empty; one in which it is mostly full, half the data is never deleted or updated, and the other half is updated cyclically; and one in which the file system is mostly full, most of the data is never updated, and a small portion is updated cyclically. The results of these endurance exper-

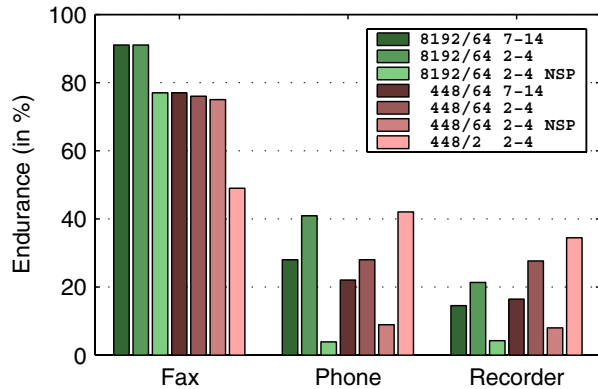


Figure 8: Endurance under different device and file-system configurations.

iments are shown in Figure 7.

The other group of experiments assesses the impact of device characteristics and file-system configuration on TFFS's performance. This group includes the same device/file-system configurations as in the capacity experiments, but the devices were kept roughly two-thirds full, with half of the data static and the other half changing cyclically. The results of this group of endurance experiments are shown in Figure 8.

The graphs show that on the fax workload, endurance is good, almost always above 75% and sometimes above 90%. On the two other workloads endurance is not as good, never reaching 50%. This is caused not by early wear of a particular block, but by a large amount of file-system structures written to the device (because writes are performed in small chunks). The endurance of the fax workload on the device with 2 KB erase units is relatively poor because fragmentation forces TFFS to erase units that are almost half empty. The other significant fact that emerges from the graphs is that the use of spare pointers significantly improves endurance (and performance, as we shall see below).

5.4 Performance Experiments

The next set of experiments is designed to measure the performance of TFFS. We measured several performance metrics under the different content scenarios (empty, full-half-static, and full-mostly-static file systems) and the different device/file-system configuration scenarios.

The first metric we measured was the average number of erasures per unit of user-data written. That is, on a device with 64 KB erase units, the number of erasures per 64 KB of user data written. The results were almost exactly the inverse of the endurance ratios (to within 0.5%). This implies that the TFFS wears out the devices almost completely evenly. When the file system performs few

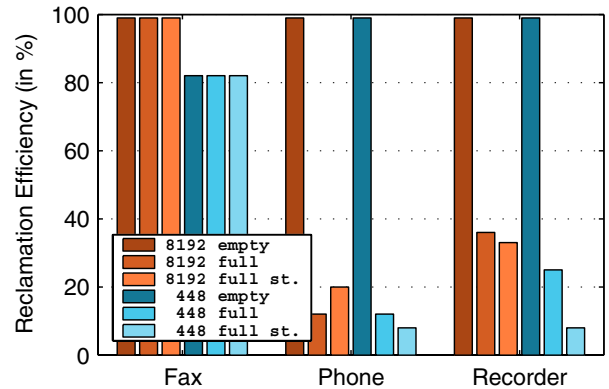


Figure 9: Reclamation efficiency under different content scenarios.

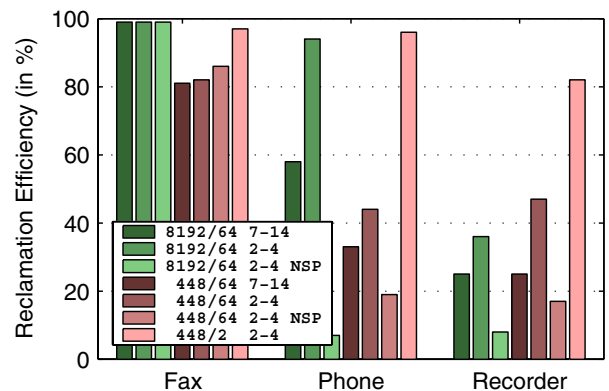


Figure 10: Reclamation efficiency under different device/file-system scenarios.

erases per unit of user data written, both performance and endurance are good. When the file system erases many units per unit of user data written, both metrics degrade. Furthermore, we have observed no cases where uneven wear leads to low endurance; low endurance is always correlated with many erasures per unit of user data written.

The second metric we measured was the efficiency of reclamations. We define this metric as the ratio of user data to the total amount of data written in block-write operations. The total amount includes writing of data to sectors, both when a sector is first created and when it is copied during reclamation, and copying of valid log entries during reclamation of the log. The denominator does not include writing of sector descriptors, erase-unit headers, and modifications of fields within sectors (fields such as spare pointers). A ratio close to 100% implies that little data is copied during reclamations, whereas a low ratio indicates that a lot of valid data is copied during reclamation. The two graphs presenting this metric,

Figure 9 and Figure 10, show that the factors that affect reclamation efficiency are primarily fullness of the file system, the amount of static data, and the size of user data items. The results again show that spare pointers contribute significantly to high performance.

We also measured two other metrics: the number of programming operations per system call and the number of flash-read instructions per system call. These metrics do not count programming and read operations performed in the context of copying blocks; these are counted by the reclamation-efficiency metric. These metrics did not reveal any interesting behavior rather than to show (again) that spare pointers improve performance. Spare pointers improve these metrics by more than a factor of 2.

6 Summary

We have presented several contributions to the area of embedded file system, especially file systems for memory-mapped flash devices.

Some of our design goals are novel. We have argued that even embedded file systems need to be recoverable (journaled), and that they should support general transactions, in order to help programmers construct reliable applications. We have also argued that in many cases, embedded file systems do not need to offer full POSIX or Windows semantics and that supporting these general semantics is expensive. The other design goals that we have attempted to achieve, such as high performance and endurance, are, of course, not novel.

We have shown that supporting recoverability and transactions is not particularly expensive. Only time will tell whether these features will actually lead to more reliable systems, but we believe that the anecdotal evidence that we presented in Section 3 (the unreliable modem/router) is typical and that due to lack of file-system support for atomicity, many embedded systems are unreliable. We have not measured the cost of supporting more general semantics.

Another area of significant contribution is the design of the data structures that TFFS uses, especially the design of the pruned versioned B-trees. This design borrows ideas from both research on persistent data structures [18, 19] and from earlier flash file systems. We have adapted the previously-proposed persistent search trees to our needs: our trees can cluster many operations on a single tree into a single version, and our algorithms prune old versions from the trees. Spare pointers are related to replacement pointers that were used in the notoriously-inefficient Microsoft Flash File System [16, 21, 22, 23, 24, 25], and to replacement block maps in the Flash Translation Layer [26, 27, 28]. But again, we have adapted these ideas: in Microsoft's FFS,

paths of replacement pointers grew and grew; in TFFS, spare pointers never increase the length of paths. The replacement blocks in the Flash Translation Layer are designed for patching elements in a table, whereas our replacement pointers are designed for a pointer-based data structure.

The performance metrics that we used to evaluate the file system are also innovative. To the best of our knowledge, most of them have never been used in the literature. The characteristics of flash are different from those of other persistent storage devices, such as traditional EEPROM and magnetic disks. Therefore, flash-specific metrics are required in order to assess and compare flash file systems. The metrics that we have introduced, such as endurance metrics and metrics that emphasize writes over reads, allow both users and file-system implementers to assess and compare file systems. We expect that additional research will utilize these metrics, perhaps even to quantitatively show that future file systems are better than TFFS.

Finally, our performance results show that TFFS does achieve its design goals, but they also point out to weaknesses. On devices with many erase units, TFFS performs very well, but it does not perform well on devices with very few erase units. This issue can be addressed either by avoiding devices with few units in TFFS-based systems, or by improving TFFS to better exploit such devices. Also, like other flash management software that manages small chunks of data on large erase units, TFFS performs poorly when the device is nearly full most of the time and contains a lot of static data.

We conjecture that some of the weaknesses in TFFS can be addressed by better file-system policies, perhaps coupled with a re-clustering mechanism. In particular, it is likely that a better allocation policy (on which erase unit to allocate sectors) and a better reclamation policy (which erase unit to reclaim) would improve endurance and performance. A re-clustering mechanism would allow TFFS to copy sectors from an erase unit being reclaimed into two or more other units; currently, the structure of logical pointers does not allow re-clustering. The allocation, reclamation, and re-clustering policies that were developed in two contexts might be applicable to TFFS. Such policies have been investigated for management of fixed-sized blocks on flash memories [5, 29, 30, 31], and for log-structured file system [3, 32, 33, 34]. Clearly not all of these techniques are applicable to flash file systems, but some of them might be. This remains an area for future research.

The design of TFFS brings up an important issue: file systems for NOR flash can write very efficiently small amounts of data, even if these writes are performed atomically and committed immediately. Writes to NOR flash are not performed in large blocks, so the time to per-

form a write operation to the file system can be roughly proportional to the amount of data written, even for small amounts. This observation is not entirely new: proportional write mechanisms have been used in Microsoft's FFS2 [16, 22, 23, 24, 25] and in other linked-list based flash file systems [35]. But these file systems suffered from performance problems and were not widely used [21]. More recent file systems tend to be block based, both in order to borrow ideas from disk file systems and in order to support NAND flash, in which writes are performed in blocks. TFFS shows that in NOR flash, it is possible to benefit from cheap small writes, without incurring the performance penalties of linked-list based designs. The same observation also led other researchers to propose flash-based application-specific persistent data structures [36, 37].

It is interesting to compare TFFS to Matchbox [11, 12] and ELF [8], two file systems for sensor-network nodes. Both are designed for the same hardware, sensor nodes with a NAND (page-mode) flash and up to 4 KB of RAM. Matchbox offers limited functionality (sequential reads and writes, a flat directory structure) and is not completely reliable. ELF offers more functionality, including random access and hierarchical directories. It appears to be more reliable than Matchbox, but still not completely reliable. ELF uses an in-RAM linked list to represent open files; these lists can be quite long if the file is long or has been updated repeatedly. It seems that some of the reliability and performance issues in ELF result from the use of NAND flash; we believe that NOR flash is more appropriate for file systems for such small embedded systems.

Acknowledgments Thanks to Yossi Shacham, Eli Luski, and Shay Izen for helpful discussion regarding flash hardware characteristics. Thanks to Andrea Arpaci-Dusseau of the USENIX program committee and to the three anonymous referees for numerous helpful comments and suggestions.

References

- [1] P. Cappelletti, C. Golla, P. Olivo, and E. Zaroni, *Flash Memories*. Kluwer, 1999.
- [2] N. A. Takahiro Ohnakado and, "Review of device technologies of flash memories," *IEICE Transactions on Electronics*, vol. E84-C, no. 6, pp. 724–733, 2001.
- [3] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992.
- [4] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An implementation of a log-structured file system for UNIX," in *USENIX Winter 1993 Conference Proceedings*, San Diego, California, Jan. 1993, pp. 307–326. [Online]. Available: citeseer.ist.psu.edu/seltzer93implementation.html
- [5] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proceedings of the USENIX 1995 Technical Conference*, New Orleans, Louisiana, Jan. 1995, pp. 155–164. [Online]. Available: <http://www.usenix.org/publications/library/proceedings/neworl/kawaguchi.html>
- [6] H.-J. Kim and S.-G. Lee, "An effective flash memory manager for reliable flash memory space management," *IEICE Transactions on Information and Systems*, vol. E85-D, no. 6, pp. 950–964, 2002.
- [7] K. M. J. Lofgren, R. D. Norman, G. B. Thelin, and A. Gupta, "Wear leveling techniques for flash EEPROM systems," U.S. Patent 6 081 447, June 27, 2000.
- [8] H. Dai, M. Neufeld, and R. Han, "ELF: An efficient log-structured flash file system for wireless micro sensor nodes," in *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Nov. 2004, pp. 176–187.
- [9] D. Woodhouse, "JFFS: The journaling flash file system," July 2001, presented in the Ottawa Linux Symposium, July 2001 (no proceedings); a 12-page article available online from <http://sources.redhat.com/jffs2/jffs2.pdf>.
- [10] (2002) YAFFS: Yet another flash filing system. Aleph One. Cambridge, UK. [Online]. Available: <http://www.aleph1.co.uk/yaffs/index.html>
- [11] D. Gay, "Design of matchbox, the simple filing system for motes (version 1.0)," Aug. 2003, available online from <http://www.tinyos.net/tinyos-1.x/doc/>.
- [12] —, "Matchbox: A simple filing system for motes (version 1.0)," Aug. 2003, available online from <http://www.tinyos.net/tinyos-1.x/doc/>.
- [13] B. Liskov and R. Rodrigues, "Transactional file systems can be fast," in *11th ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.
- [14] M. Seltzer and M. Stonebraker, "Transaction support in read optimized and write optimized file systems," in *Proceedings of the 16th Conference on Very Large Databases*, Brisbane, 1990.

- [15] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," July 2004, submitted to the *ACM Computing Surveys*.
- [16] P. Torelli, "The Microsoft flash file system," *Dr. Dobbs's Journal*, pp. 62–72, Feb. 1995. [Online]. Available: <http://www.ddj.com>
- [17] S. Wells, R. N. Hasbun, and K. Robinson, "Sector-based storage device emulator having variable-sized sector," U.S. Patent 5 822 781, Oct. 13, 1998.
- [18] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *Journal of Computer and System Sciences*, vol. 38, no. 1, pp. 86–124, 1989.
- [19] H. Kaplan, "Persistent data structures," in *Handbook of Data Structures and Applications*, D. P. Mehta and S. Sahni, Eds. CRC Press, 2004.
- [20] D. Hitz, J. Lau, , and M. Malcolm, "File system design for an NFS file server appliance," in *Proceedings of the 1994 Winter USENIX Technical Conference*, San Francisco, CA, Jan. 1994, pp. 235–245.
- [21] F. Dougliis, R. Caceres, M. F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage alternatives for mobile computers," in *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, California, Nov. 1994, pp. 25–37. [Online]. Available: citeseer.ist.psu.edu/dougliis94storage.html
- [22] P. L. Barrett, S. D. Quinn, and R. A. Lipe, "System for updating data stored on a flash-erasable, programmable, read-only memory (FEPROM) based upon predetermined bit value of indicating pointers," U.S. Patent 5 392 427, Feb. 21, 1995.
- [23] W. J. Krueger and S. Rajagopalan, "Method and system for file system management using a flash-erasable, programmable, read-only memory," U.S. Patent 5 634 050, May 27, 1999.
- [24] —, "Method and system for file system management using a flash-erasable, programmable, read-only memory," U.S. Patent 5 898 868, Apr. 27, 1999.
- [25] —, "Method and system for file system management using a flash-erasable, programmable, read-only memory," U.S. Patent 6 256 642, July 3, 2001.
- [26] A. Ban, "Flash file system," U.S. Patent 5 404 485, Apr. 4, 1995.
- [27] —, "Flash file system optimized for page-mode flash technologies," U.S. Patent 5 937 425, Aug. 10, 1999.
- [28] Intel Corporation, "Understanding the flash translation layer (FTL) specification," Application Note 648, 1998.
- [29] M.-L. Chiang and R.-C. Chang, "Cleaning policies in mobile computers using flash memory," *The Journal of Systems and Software*, vol. 48, no. 3, pp. 213–231, 1999.
- [30] M.-L. Chiang, P. C. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Software—Practice and Experience*, vol. 29, no. 3, 1999.
- [31] M. Wu and W. Zwaenepoel, "eNVy: a non-volatile, main memory storage system," in *Proceedings of the 6th international conference on Architectural support for programming languages and operating systems*. ACM Press, 1994, pp. 86–97.
- [32] T. Blackwell, J. Harris, and M. Seltzer, "Heuristic cleaning algorithms in log-structured file systems," in *Proceedings of the 1995 USENIX Technical Conference*, Jan. 1995, pp. 277–288.
- [33] J. Wang and Y. Hu, "A novel reordering write buffer to improve write performance of log-structured file systems," *IEEE Transactions on Computers*, vol. 52, no. 12, pp. 1559–1572, Dec. 2003.
- [34] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson, "Improving the performance of log-structured file systems with adaptive methods," in *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, Oct. 1997, pp. 238–251.
- [35] N. Daberko, "Operating system including improved file management for use in devices utilizing flash memory as main memory," U.S. Patent 5 787 445, July 28, 1998.
- [36] C.-H. Wu, L.-P. Chang, and T.-W. Kuo, "An efficient B-tree layer for flash-memory storage systems," in *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, Tainan City, Taiwan, Feb. 2003, 20 pages.
- [37] —, "An efficient R-tree implementation over flash-memory storage systems," in *Proceedings of the eleventh ACM international symposium on Advances in geographic information systems*. ACM Press, 2003, pp. 17–24.

Analysis and Evolution of Journaling File Systems

Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

*Computer Sciences Department
University of Wisconsin, Madison
{vijayan, dusseau, remzi}@cs.wisc.edu*

Abstract

We develop and apply two new methods for analyzing file system behavior and evaluating file system changes. First, *semantic block-level analysis (SBA)* combines knowledge of on-disk data structures with a trace of disk traffic to infer file system behavior; in contrast to standard benchmarking approaches, SBA enables users to understand *why* the file system behaves as it does. Second, *semantic trace playback (STP)* enables traces of disk traffic to be easily modified to represent changes in the file system implementation; in contrast to directly modifying the file system, STP enables users to rapidly gauge the benefits of new policies. We use SBA to analyze Linux ext3, ReiserFS, JFS, and Windows NTFS; in the process, we uncover many strengths and weaknesses of these journaling file systems. We also apply STP to evaluate several modifications to ext3, demonstrating the benefits of various optimizations without incurring the costs of a real implementation.

1 Introduction

Modern file systems are journaling file systems [4, 22, 29, 32]. By writing information about pending updates to a write-ahead log [12] before committing the updates to disk, journaling enables fast file system recovery after a crash. Although the basic techniques have existed for many years (*e.g.*, in Cedar [13] and Episode [9]), journaling has increased in popularity and importance in recent years; due to ever-increasing disk capacities, scan-based recovery (*e.g.*, via fsck [16]) is prohibitively slow on modern drives and RAID volumes. However, despite the popularity and importance of journaling file systems such as ext3 [32], ReiserFS [22], JFS [4], and NTFS [27] little is known about their internal policies.

Understanding how these file systems behave is important for developers, administrators, and application writers. Therefore, we believe it is time to perform a detailed analysis of journaling file systems. Most previous work has analyzed file systems from above; by writing user-level programs and measuring the time taken for various file system operations, one can elicit some salient aspects of file system performance [6, 8, 19, 26]. However, it is difficult to discover the underlying reasons for the observed performance with this approach.

In this paper we employ a novel benchmarking methodology called *semantic block-level analysis (SBA)* to trace and analyze file systems. With SBA, we induce controlled

workload patterns from above the file system, but focus our analysis not only on the time taken for said operations, but also on the resulting stream of read and write requests *below* the file system. This analysis is *semantic* because we leverage information about block type (*e.g.*, whether a block request is to the journal or to an inode); this analysis is *block-level* because it interposes on the block interface to storage. By analyzing the low-level block stream in a semantically meaningful way, one can understand *why* the file system behaves as it does.

Analysis hints at how the file system could be improved, but does not reveal whether the change is worth implementing. Traditionally, for each potential improvement to the file system, one must implement the change and measure performance under various workloads; if the change gives little improvement, the implementation effort is wasted. In this paper, we introduce and apply a complementary technique to SBA called *semantic trace playback (STP)*. STP enables us to rapidly suggest and evaluate file system modifications without a large implementation or simulation effort. Using real workloads and traces, we show how STP can be used effectively.

We have applied a detailed analysis to both Linux ext3 and ReiserFS and a preliminary analysis to Linux JFS and Windows NTFS. In each case, we focus on the journaling aspects of each file system. For example, we determine the events that cause data and metadata to be written to the journal or their fixed locations. We also examine how the characteristics of the workload and configuration parameters (*e.g.*, the size of the journal and the values of commit timers) impact this behavior.

Our analysis has uncovered design flaws, performance problems, and even correctness bugs in these file systems. For example, ext3 and ReiserFS make the design decision to group unrelated traffic into the same compound transaction; the result of this *tangled synchrony* is that a single disk-intensive process forces *all* write traffic to disk, particularly affecting the performance of otherwise asynchronous writers. (§3.2.1). Further, we find that both ext3 and ReiserFS artificially *limit parallelism*, by preventing the overlap of pre-commit journal writes and fixed-place updates (§3.2.2). Our analysis also reveals that in ordered and data journaling modes, ext3 exhibits *eager writing*, forcing data blocks to disk much sooner than the typical 30-second delay (§3.2.3). In addition, we find that JFS

has an *infinite write delay*, as it does not utilize commit timers and indefinitely postpones journal writes until another trigger forces writes to occur, such as memory pressure (§5). Finally, we identify four previously unknown bugs in ReiserFS that will be fixed in subsequent releases (§4.3).

The main contributions of this paper are:

- A new methodology, semantic block analysis (SBA), for understanding the internal behavior of file systems.
- A new methodology, semantic trace playback (STP), for rapidly gauging the benefits of file system modifications without a heavy implementation effort.
- A detailed analysis using SBA of two important journaling file systems, ext3 and ReiserFS, and a preliminary analysis of JFS and NTFS.
- An evaluation using STP of different design and implementation alternatives for ext3.

The rest of this paper is organized as follows. In §2 we describe our new techniques for SBA and STP. We apply these techniques to ext3, ReiserFS, JFS, and NTFS in §3, §4, §5, and §6 respectively. We discuss related work in §7 and conclude in §8.

2 Methodology

We introduce two techniques for evaluating file systems. First, semantic block analysis (SBA) enables users to understand the internal behavior and policies of the file system. Second, semantic trace playback (STP) allows users to quantify how changing the file system will impact the performance of real workloads.

2.1 Semantic Block-Level Analysis

File systems have traditionally been evaluated using one of two approaches; either one applies synthetic or real workloads and measures the resulting file system performance [6, 14, 17, 19, 20] or one collects traces to understand how file systems are used [1, 2, 21, 24, 35, 37]. However, performing each in isolation misses an interesting opportunity: by correlating the observed disk traffic with the running workload and with performance, one can often answer *why* a given workload behaves as it does.

Block-level tracing of disk traffic allows one to analyze a number of interesting properties of the file system and workload. At the coarsest granularity, one can record the *quantity* of disk traffic and how it is divided between reads and writes; for example, such information is useful for understanding how file system caching and write buffering affect performance. At a more detailed level, one can track the *block number* of each block that is read or written; by analyzing the block numbers, one can see the extent to which traffic is sequential or random. Finally, one can analyze the *timing* of each block; with timing information, one can understand when the file system initiates a burst of traffic.

By combining block-level analysis with *semantic* information about those blocks, one can infer much more about

	Ext3	ReiserFS	JFS	NTFS
SBA Generic	1289	1289	1289	1289
SBA FS Specific	181	48	20	15
SBA Total	1470	1337	1309	1304

Table 1: **Code size of SBA drivers.** *The number of C statements (counted as the number of semicolons) needed to implement SBA for ext3 and ReiserFS and a preliminary SBA for JFS and NTFS.*

the behavior of the file system. The main difference between *semantic block analysis* (SBA) and more standard block-level tracing is that SBA analysis understands the on-disk format of the file system under test. SBA enables us to understand new properties of the file system. For example, SBA allows us to distinguish between traffic to the journal versus to in-place data and to even track individual transactions to the journal.

2.1.1 Implementation

The infrastructure for performing SBA is straightforward. One places a pseudo-device driver in the kernel, associates it with an underlying disk, and mounts the file system of interest (*e.g.*, ext3) on the pseudo device; we refer to this as the SBA driver. One then runs controlled microbenchmarks to generate disk traffic. As the SBA driver passes the traffic to and from the disk, it also efficiently tracks each request and response by storing a small record in a fixed-sized circular buffer. Note that by tracking the ordering of requests and responses, the pseudo-device driver can infer the order in which the requests were scheduled at lower levels of the system.

SBA requires that one interpret the *contents* of the disk block traffic. For example, one must interpret the contents of the journal to infer the type of journal block (*e.g.*, a descriptor or commit block) and one must interpret the journal descriptor block to know which data blocks are journaled. As a result, it is most efficient to semantically interpret block-level traces on-line; performing this analysis off-line would require exporting the contents of blocks, greatly inflating the size of the trace.

An SBA driver is customized to the file system under test. One concern is the amount of information that must be embedded within the SBA driver for each file system. Given that the focus of this paper is on understanding journaling file systems, our SBA drivers are embedded with enough information to interpret the placement and contents of journal blocks, metadata, and data blocks. We now analyze the complexity of the SBA driver for four journaling file systems, ext3, ReiserFS, JFS, and NTFS.

Journaling file systems have both a journal, where transactions are temporarily recorded, and fixed-location data structures, where data permanently reside. Our SBA driver distinguishes between the traffic sent to the journal and to the fixed-location data structures. This traffic is simple to distinguish in ReiserFS, JFS, and NTFS because the journal is a set of contiguous blocks, separate from the rest of the file system. However, to be backward

compatible with ext2, ext3 can treat the journal as a regular file. Thus, to determine which blocks belong to the journal, SBA uses its knowledge of inodes and indirect blocks; given that the journal does not change location after it has been created, this classification remains efficient at run-time. SBA is also able to classify the different types of journal blocks such as the descriptor block, journal data block, and commit block.

To perform useful analysis of journaling file systems, the SBA driver does not have to understand many details of the file system. For example, our driver does not understand the directory blocks or superblock of ext3 or the B+ tree structure of ReiserFS or JFS. However, if one wishes to infer additional file system properties, one may need to embed the SBA driver with more knowledge. Nevertheless, the SBA driver does not know anything about the policies or parameters of the file system; in fact, SBA can be used to infer these policies and parameters.

Table 1 reports the number of C statements required to implement the SBA driver. These numbers show that most of the code in the SBA driver (*i.e.*, 1289 statements) is for general infrastructure; only between approximately 50 and 200 statements are needed to support different journaling file systems. The ext3 specific code is more than that of the other file systems because in ext3, journal is created as a file and can span multiple block groups. In order to find the blocks belonging to the journal file, we parse the journal inode and journal indirect blocks. In Reiserfs, JFS, and NTFS the journal is contiguous and finding its blocks is trivial (even though the journal is a file in NTFS, for small journals they are contiguously allocated).

2.1.2 Workloads

SBA analysis can be used to gather useful information for any workload. However, the focus of this paper is on understanding the internal policies and behavior of the file system. As a result, we wish to construct synthetic workloads that uncover decisions made by the file system. More realistic workloads will be considered only when we apply semantic trace playback.

When constructing synthetic workloads that stress the file system, previous research has revealed a range of parameters that impact performance [8]. We have created synthetic workloads varying these parameters: the amount of data written, sequential versus random accesses, the interval between calls to `fsync`, and the amount of concurrency. We focus exclusively on write-based workloads because reads are directed to their fixed-place location, and thus do not impact the journal. When we analyze each file system, we only report results for those workloads which revealed file system policies and parameters.

2.1.3 Overhead of SBA

The processing and memory overheads of SBA are minimal for the workloads we ran as they did not generate high

I/O rates. For every I/O request, the SBA driver performs the following operations to collect detailed traces:

- A `gettimeofday()` call during the start and end of I/O.
- A block number comparison to see if the block is a journal or fixed-location block.
- A check for a magic number on journal blocks to distinguish journal metadata from journal data.

SBA stores the trace records with details like read or write, block number, block type, time of issue and completion in an internal circular buffer. All these operations are performed only if one needs detailed traces. But for many of our analyses, it is sufficient to have cumulative statistics like the total number of journal writes and fixed-location writes. These numbers are easy to collect and require less processing within the SBA driver.

2.1.4 Alternative Approaches

One might believe that directly instrumenting a file system to obtain timing information and disk traces would be equivalent or superior to performing SBA analysis. We believe this is not the case for several reasons. First, to directly instrument the file system, one needs source code for that file system and one must re-instrument new versions as they are released; in contrast, SBA analysis does not require file system source and much of the SBA driver code can be reused across file systems and versions. Second, when directly instrumenting the file system, one may accidentally miss some of the conditions for which disk blocks are written; however, the SBA driver is guaranteed to see all disk traffic. Finally, instrumenting existing code may accidentally change the behavior of that code [36]; an efficient SBA driver will likely have no impact on file system behavior.

2.2 Semantic Trace Playback

In this section we describe semantic trace playback (STP). STP can be used to rapidly evaluate certain kinds of new file system designs, both without a heavy implementation investment and without a detailed file system simulator.

We now describe how STP functions. STP is built as a user-level process; it takes as input a trace (described further below), parses it, and issues I/O requests to the disk using the raw disk interface. Multiple threads are employed to allow for concurrency.

Ideally, STP would function by only taking a block-level trace as input (generated by the SBA driver), and indeed this is sufficient for some types of file system modifications. For example, it is straightforward to model different layout schemes by simply mapping blocks to different on-disk locations.

However, it was our desire to enable more powerful emulations with STP. For example, one issue we explore later is the effect of using byte differences in the journal, instead of storing entire blocks therein. One complication that arises is that by changing the contents of the journal,

the *timing* of block I/O changes; the thresholds that initiate I/O are triggered at a different time.

To handle emulations that alter the timing of disk I/O, more information is needed than is readily available in the low-level block trace. Specifically, STP needs to observe two high-level activities. First, STP needs to observe any file-system level operations that create dirty buffers in memory. The reason for this requirement is found in §3.2.2; when the number of uncommitted buffers reaches a threshold (in ext3, $\frac{1}{4}$ of the journal size), a commit is enacted. Similarly, when one of the interval timers expires, these blocks may have to be flushed to disk.

Second, STP needs to observe application-level calls to `fsync`; without doing so, STP cannot understand whether an I/O operation in the SBA trace is there due to a `fsync` call or due to normal file system behavior (*e.g.*, thresholds being crossed, timers going off, etc.). Without such differentiation, STP cannot emulate behaviors that are timing sensitive.

Both of these requirements are met by giving a file-system level trace as input to STP, in addition to the SBA-generated block-level trace. We currently use library-level interpositioning to trace the application of interest.

We can now qualitatively compare STP to two other standard approaches for file system evolution. In the first approach, when one has an idea for improving a file system, one simply implements the idea within the file system and measures the performance of the real system. This approach is attractive because it gives a reliable answer as to whether the idea was a real improvement, assuming that the workload applied is relevant. However, it is time consuming, particularly if the modification to the file system is non-trivial.

In the second approach, one builds an accurate simulation of the file system, and evaluates a new idea within the domain of the file system before migrating it to the real system. This approach is attractive because one can often avoid some of the details of building a real implementation and thus more quickly understand whether the idea is a good one. However, it requires a detailed and accurate simulator, the construction and maintenance of which is certainly a challenging endeavor.

STP avoids the difficulties of both of these approaches by using the low-level traces as the “truth” about how the file system behaves, and then modifying file system output (*i.e.*, the block stream) based on its simple internal models of file system behavior; these models are based on our empirical analysis found in §3.2.

Despite its advantages over traditional implementation and simulation, STP is limited in some important ways. For example, STP is best suited for evaluating design alternatives under simpler benchmarks; if the workload exhibits complex virtual memory behavior whose interactions with the file system are not modeled, the results may not be meaningful. Also, STP is limited to evaluating file system changes that are not too radical; the basic opera-

tion of the file system should remain intact. Finally, STP does not provide a means to evaluate *how* to implement a given change; rather, it should be used to understand *whether* a certain modification improves performance.

2.3 Environment

All measurements are taken on a machine running Linux 2.4.18 with a 600 MHz Pentium III processor and 1 GB of main memory. The file system under test is created on a single IBM 9LZX disk, which is separate from the root disk. Where appropriate, each data point reports the average of 30 trials; in all cases, variance is quite low.

3 The Ext3 File System

In this section, we analyze the popular Linux filesystem, ext3. We begin by giving a brief overview of ext3, and then apply semantic block-level analysis and semantic trace playback to understand its internal behavior.

3.1 Background

Linux ext3 [33, 34] is a journaling file system, built as an extension to the ext2 file system. In ext3, data and meta-data are eventually placed into the standard ext2 structures, which are the fixed-location structures. In this organization (which is loosely based on FFS [15]), the disk is split into a number of *block groups*; within each block group are bitmaps, inode blocks, and data blocks. The ext3 journal (or log) is commonly stored as a file within the file system, although it can be stored on a separate device or partition. Figure 1 depicts the ext3 on-disk layout.

Information about pending file system updates is written to the journal. By forcing journal updates to disk *before* updating complex file system structures, this write-ahead logging technique [12] enables efficient crash recovery; a simple scan of the journal and a redo of any incomplete committed operations bring the file system to a consistent state. During normal operation, the journal is treated as a circular buffer; once the necessary information has been propagated to its fixed location in the ext2 structures, journal space can be reclaimed.

Journaling Modes: Linux ext3 includes three flavors of journaling: *writeback mode*, *ordered mode*, and *data journaling mode*; Figure 2 illustrates the differences between these modes. The choice of mode is made at mount time and can be changed via a remount.

In *writeback mode*, only file system metadata is journaled; data blocks are written directly to their fixed location. This mode does not enforce any ordering between the journal and fixed-location data writes, and because of this, writeback mode has the weakest consistency semantics of the three modes. Although it guarantees consistent file system metadata, it does not provide any guarantee as to the consistency of data blocks.

In *ordered journaling mode*, again only metadata writes are journaled; however, data writes to their fixed location are ordered *before* the journal writes of the metadata. In



IB = Inode Bitmap, DB = Data Bitmap, JS = Journal Superblock, JD = Journal Descriptor Block, JC = Journal Commit Block

Figure 1: **Ext3 On-Disk Layout.** The picture shows the layout of an ext3 file system. The disk address space is broken down into a series of block groups (akin to FFS cylinder groups), each of which has bitmaps to track allocations and regions for inodes and data blocks. The ext3 journal is depicted here as a file within the first block group of the file system; it contains a superbloc, various descriptor blocks to describe its contents, and commit blocks to denote the ends of transactions.

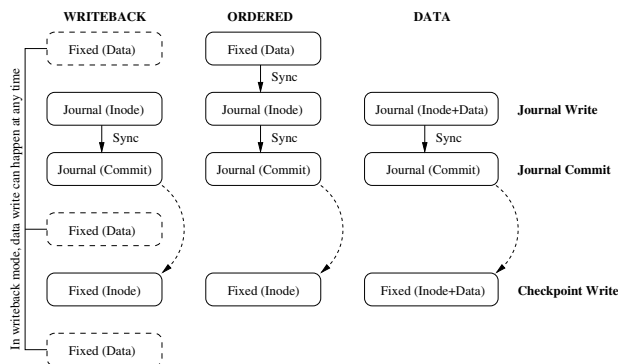


Figure 2: **Ext3 Journaling Modes.** The diagram depicts the three different journaling modes of ext3: writeback, ordered, and data. In the diagram, time flows downward. Boxes represent updates to the file system, e.g., “Journal (Inode)” implies the write of an inode to the journal; the other destination for writes is labeled “Fixed”, which is a write to the fixed in-place ext2 structures. An arrow labeled with a “Sync” implies that the two blocks are written out in immediate succession synchronously, hence guaranteeing the first completes before the second. A curved arrow indicates ordering but not immediate succession; hence, the second write will happen at some later time. Finally, for writeback mode, the dashed box around the “Fixed (Data)” block indicates that it may happen at any time in the sequence. In this example, we consider a data block write and its inode as the updates that need to be propagated to the file system; the diagrams show how the data flow is different for each of the ext3 journaling modes.

contrast to writeback mode, this mode provides more sensible consistency semantics, where both the data and the metadata are guaranteed to be consistent after recovery.

In full data journaling mode, ext3 logs both metadata and data to the journal. This decision implies that when a process writes a data block, it will typically be written out to disk *twice*: once to the journal, and then later to its fixed ext2 location. Data journaling mode provides the same strong consistency guarantees as ordered journaling mode; however, it has different performance characteristics, in some cases worse, and surprisingly, in some cases, better. We explore this topic further (§3.2).

Transactions: Instead of considering each file system update as a separate transaction, ext3 groups many updates into a single *compound transaction* that is periodically committed to disk. This approach is relatively simple to implement [33]. Compound transactions may have better performance than more fine-grained transactions when the same structure is frequently updated in a short period of time (e.g., a free space bitmap or an inode of a file that

is constantly being extended) [13].

Journal Structure: Ext3 uses additional metadata structures to track the list of journaled blocks. The *journal superbloc* tracks summary information for the journal, such as the block size and head and tail pointers. A *journal descriptor block* marks the beginning of a transaction and describes the subsequent journaled blocks, including their final fixed on-disk location. In data journaling mode, the descriptor block is followed by the data and metadata blocks; in ordered and writeback mode, the descriptor block is followed by the metadata blocks. In all modes, ext3 logs full blocks, as opposed to differences from old versions; thus, even a single bit change in a bitmap results in the entire bitmap block being logged. Depending upon the size of the transaction, multiple descriptor blocks each followed by the corresponding data and metadata blocks may be logged. Finally, a *journal commit block* is written to the journal at the end of the transaction; once the commit block is written, the journaled data can be recovered without loss.

Checkpointing: The process of writing journaled metadata and data to their fixed-locations is known as *checkpointing*. Checkpointing is triggered when various thresholds are crossed, e.g., when file system buffer space is low, when there is little free space left in the journal, or when a timer expires.

Crash Recovery: Crash recovery is straightforward in ext3 (as it is in many journaling file systems); a basic form of *redo logging* is used. Because new updates (whether to data or just metadata) are written to the log, the process of restoring in-place file system structures is easy. During recovery, the file system scans the log for committed complete transactions; incomplete transactions are discarded. Each update in a completed transaction is simply replayed into the fixed-place ext2 structures.

3.2 Analysis of ext3 with SBA

We now perform a detailed analysis of ext3 using our SBA framework. Our analysis is divided into three categories. First, we analyze the basic behavior of ext3 as a function of the workload and the three journaling modes. Second, we isolate the factors that control when data is committed to the journal. Third, we isolate the factors that control when data is checkpointed to its fixed-place location.

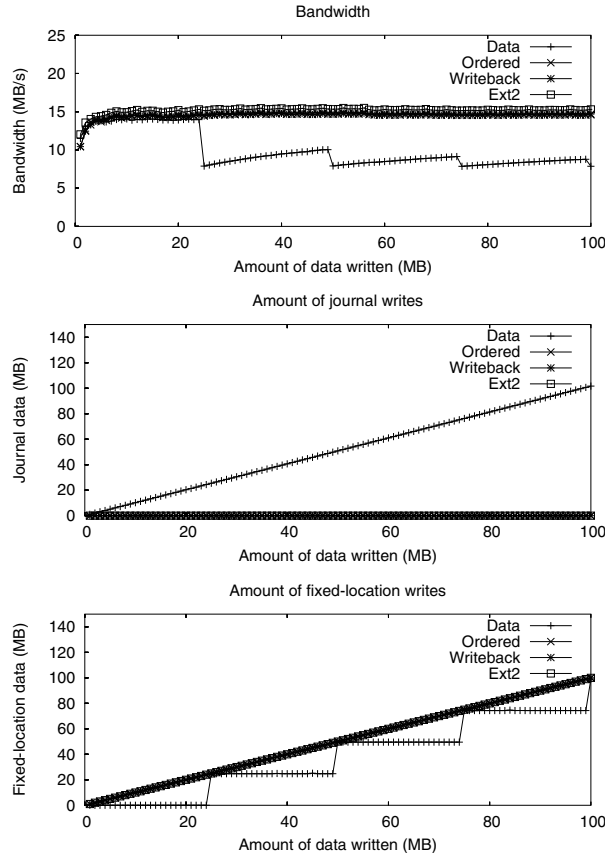


Figure 3: **Basic Behavior for Sequential Workloads in ext3.** Within each graph, we evaluate ext2 and the three ext3 journaling modes. We increase the size of the written file along the x-axis. The workload writes to a single file sequentially and then performs an `fsync`. Each graph examines a different metric: the top graph shows the achieved bandwidth; the middle graph uses SBA to report the amount of journal traffic; the bottom graph uses SBA to report the amount of fixed-location traffic. The journal size is set to 50 MB.

3.2.1 Basic Behavior: Modes and Workload

We begin by analyzing the basic behavior of ext3 as a function of the workload and journaling mode (*i.e.*, writeback, ordered, and full data journaling). Our goal is to understand the workload conditions that trigger ext3 to write data and metadata to the journal and to their fixed locations. We explored a range of workloads by varying the amount of data written, the sequentiality of the writes, the synchronization interval between writes, and the number of concurrent writers.

Sequential and Random Workloads: We begin by showing our results for three basic workloads. The first workload writes to a single file sequentially and then performs an `fsync` to flush its data to disk (Figure 3); the second workload issues 4 KB writes to random locations in a single file and calls `fsync` once for every 256 writes (Figure 4); the third workload again issues 4 KB random writes but calls `fsync` for every write (Figure 5). In each workload, we increase the total amount of data that

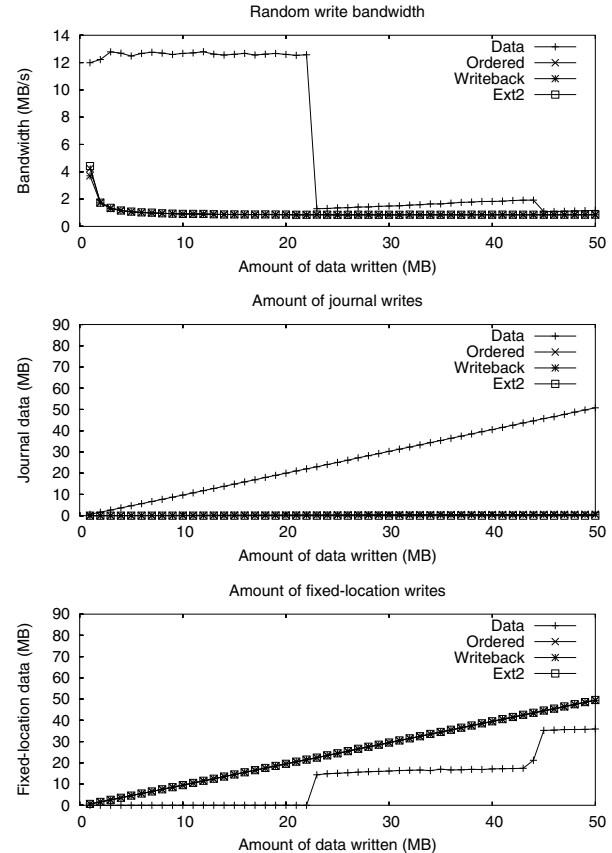


Figure 4: **Basic Behavior for Random Workloads in ext3.** This figure is similar to Figure 3. The workload issues 4 KB writes to random locations in a single file and calls `fsync` once for every 256 writes. Top graph shows the bandwidth, middle graph shows the journal traffic, and the bottom graph reports the fixed-location traffic. The journal size is set to 50 MB.

it writes and observe how the behavior of ext3 changes.

The top graphs in Figures 3, 4, and 5 plot the achieved bandwidth for the three workloads; within each graph, we compare the three different journaling modes and ext2. From these bandwidth graphs we make four observations. First, the achieved bandwidth is extremely sensitive to the workload: as expected, a sequential workload achieves much higher throughput than a random workload and calling `fsync` more frequently further reduces throughput for random workloads. Second, for sequential traffic, ext2 performs slightly better than the highest performing ext3 mode: there is a small but noticeable cost to journaling for sequential streams. Third, for all workloads, ordered mode and writeback mode achieve bandwidths that are similar to ext2. Finally, the performance of data journaling is quite irregular, varying in a sawtooth pattern with the amount of data written.

These graphs of file system throughput allow us to compare performance across workloads and journaling modes, but do not enable us to infer the *cause* of the differences. To help us infer the internal behavior of the file system, we apply semantic analysis to the underlying block stream;

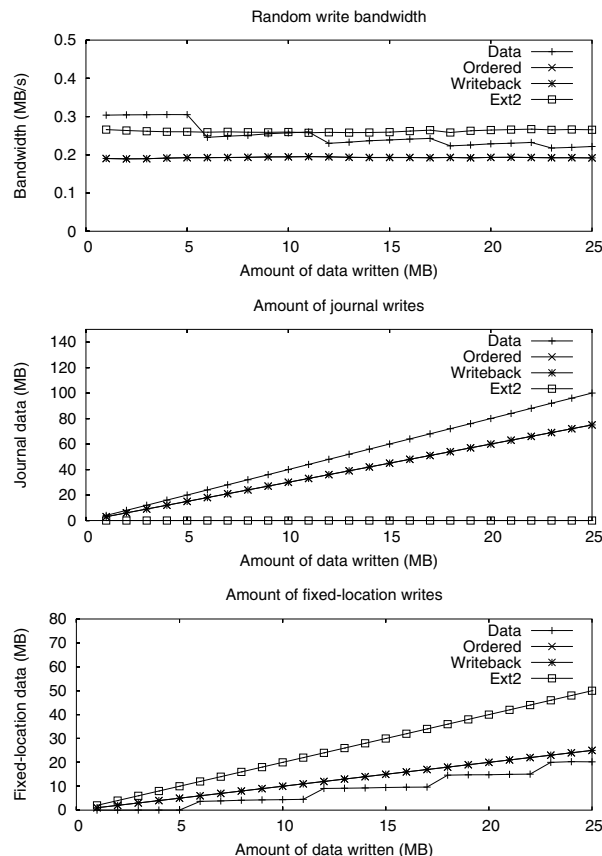


Figure 5: **Basic Behavior for Random Workloads in ext3.** This figure is similar to Figure 3. The workload issues 4 KB random writes and calls `fsync` for every write. Bandwidth is shown in the first graph; journal writes and fixed-location writes are reported in the second and third graph using SBA. The journal size is set to 50 MB.

in particular, we record the amount of journal and fixed-location traffic. This accounting is shown in the bottom two graphs of Figures 3, 4, and 5.

The second row of graphs in Figures 3, 4, and 5 quantify the amount of traffic flushed to the journal and help us to infer the events which cause this traffic. We see that, in data journaling mode, the total amount of data written to the journal is high, proportional to the amount of data written by the application; this is as expected, since both data and metadata are journaled. In the other two modes, only metadata is journaled; therefore, the amount of traffic to the journal is quite small.

The third row of Figures 3, 4, and 5 shows the traffic to the fixed location. For writeback and ordered mode the amount of traffic written to the fixed location is equal to the amount of data written by the application. However, in data journaling mode, we observe a stair-stepped pattern in the amount of data written to the fixed location. For example, with a file size of 20 MB, even though the process has called `fsync` to force the data to disk, no data is written to the fixed location by the time the application terminates; because all data is logged, the expected

consistency semantics are still preserved. However, even though it is not necessary for consistency, when the application writes more data, checkpointing does occur at regular intervals; this extra traffic leads to the sawtooth bandwidth measured in the first graph. In this particular experiment with sequential traffic and a journal size of 50 MB, a checkpoint occurs when 25 MB of data is written; we explore the relationship between checkpoints and journal size more carefully in §3.2.3.

The SBA graphs also reveal why data journaling mode performs better than the other modes for asynchronous random writes. With data journaling mode, all data is written first to the log, and thus even random writes become *logically* sequential and achieve sequential bandwidth. As the journal is filled, checkpointing causes extra disk traffic, which reduces bandwidth; in this particular experiment, the checkpointing occurs near 23 MB. Finally, SBA analysis reveals that synchronous 4 KB writes do not perform well, even in data journaling mode. Forcing each small 4 KB write to the log, even in logical sequence, incurs a delay between sequential writes (not shown) and thus each write incurs a disk rotation.

Concurrency: We now report our results from running workloads containing multiple processes. We construct a workload containing two diverse classes of traffic: an asynchronous foreground process in competition with a background process. The foreground process writes out a 50 MB file without calling `fsync`, while the background process repeatedly writes a 4 KB block to a random location, optionally calls `fsync`, and then sleeps for some period of time (*i.e.*, the “sync interval”). We focus on data journaling mode, but the effect holds for ordered journaling mode too (not shown).

In Figure 6 we show the impact of varying the mean “sync interval” of the background process on the performance of the foreground process. The first graph plots the bandwidth achieved by the foreground asynchronous process, depending upon whether it competes against an asynchronous or synchronous background process. As expected, when the foreground process runs with an asynchronous background process, its bandwidth is uniformly high and matches in-memory speeds. However, when the foreground process competes with a synchronous background process, its bandwidth drops to disk speeds.

The SBA analysis in the second graph reports the amount of journal data, revealing that the more frequently the background process calls `fsync`, the more traffic is sent to the journal. In fact, the amount of journal traffic is equal to the sum of the foreground and background process traffic written in that interval, not that of only the background process. This effect is due to the implementation of compound transactions in ext3: all file system updates add their changes to a global transaction, which is eventually committed to disk.

This workload reveals the potentially disastrous consequences of grouping unrelated updates into the same com-

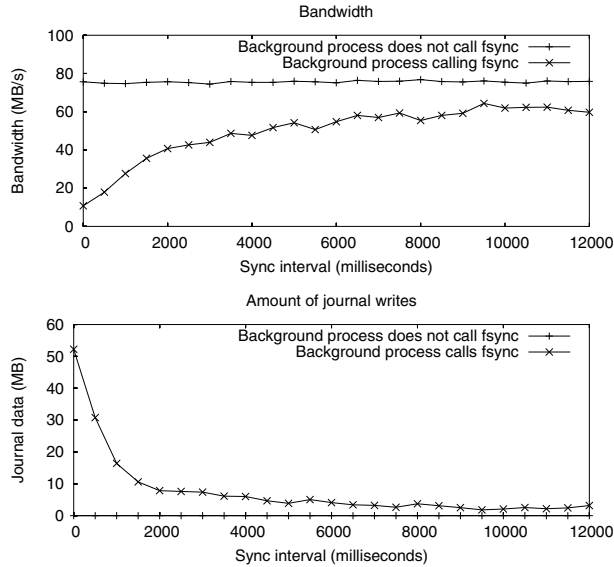


Figure 6: **Basic Behavior for Concurrent Writes in ext3.** Two processes compete in this workload: a foreground process writing a sequential file of size 50 MB and a background process writing out 4 KB, optionally calling `fsync`, sleeping for the “sync interval”, and then repeating. Along the x-axis, we increase the sync interval. In the top graph, we plot the bandwidth achieved by the foreground process in two scenarios: with the background process either calling or not calling `fsync` after each write. In the bottom graph, the amount of data written to disk during both sets of experiments is shown.

pound transaction: all traffic is committed to disk at the same rate. Thus, even asynchronous traffic must wait for synchronous updates to complete. We refer to this negative effect as *tangled synchrony* and explore the benefits of untangling transactions in §3.3.3 using STP.

3.2.2 Journal Commit Policy

We next explore the conditions under which ext3 commits transactions to its on-disk journal. As we will see, two factors influence this event: the size of the journal and the settings of the commit timers.

In these experiments, we focus on data journaling mode; since this mode writes both metadata and data to the journal, the traffic sent to the journal is most easily seen in this mode. However, writeback and ordered modes commit transactions using the same policies. To exercise log commits, we examine workloads in which data is not explicitly forced to disk by the application (*i.e.*, the process does not call `fsync`); further, to minimize the amount of metadata overhead, we write to a single file.

Impact of Journal Size: The size of the journal is a configurable parameter in ext3 that contributes to when updates are committed. By varying the size of the journal and the amount of data written in the workload, we can infer the amount of data that triggers a log commit. Figure 7 shows the resulting bandwidth and the amount of journal traffic, as a function of file size and journal size. The first graph shows that when the amount of data writ-

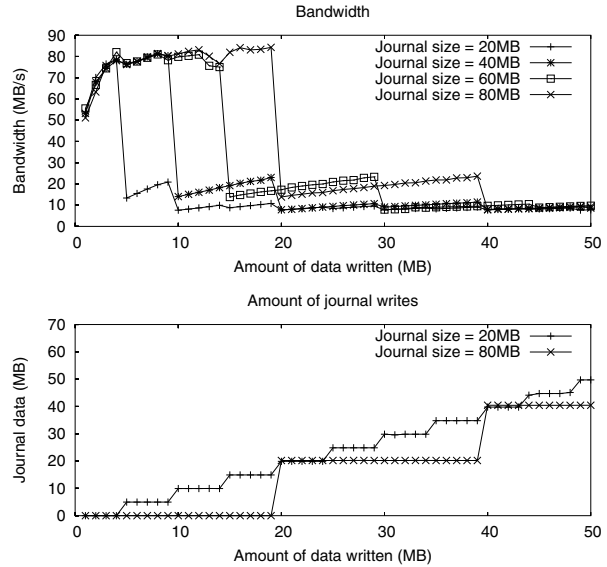


Figure 7: **Impact of Journal Size on Commit Policy in ext3.** The topmost figure plots the bandwidth of data journaling mode under different-sized file writes. Four lines are plotted representing four different journal sizes. The second graph shows the amount of log traffic generated for each of the experiments (for clarity, only two of the four journal sizes are shown).

ten by the application (to be precise, the number of dirty uncommitted buffers, which includes both data and metadata) reaches $\frac{1}{4}$ the size of the journal, bandwidth drops considerably. In fact, in the first performance regime, the observed bandwidth is equal to in-memory speeds.

Our semantic analysis, shown in the second graph, reports the amount of traffic to the journal. This graph reveals that metadata and data are forced to the journal when it is equal to $\frac{1}{4}$ the journal size. Inspection of Linux ext3 code confirms this threshold. Note that the threshold is the same for ordered and writeback modes (not shown); however, it is triggered much less frequently since only metadata is logged.

Impact of Timers: In Linux 2.4 ext3, three timers have some control over when data is written: the metadata commit timer and the data commit timer, both managed by the `kupdate` daemon, and the commit timer managed by the `kjournal` daemon. The system-wide `kupdate` daemon is responsible for flushing dirty buffers to disk; the `kjournal` daemon is specialized for ext3 and is responsible for committing ext3 transactions. The strategy for ext2 is to flush metadata frequently (*e.g.*, every 5 seconds) while delaying data writes for a longer time (*e.g.*, every 30 seconds). Flushing metadata frequently has the advantage that the file system can approach FFS-like consistency without a severe performance penalty; delaying data writes has the advantage that files that are deleted quickly do not tax the disk. Thus, mapping the ext2 goals to the ext3 timers leads to default values of 5 seconds for the `kupdate` metadata timer, 5 seconds for the `kjournal` timer,

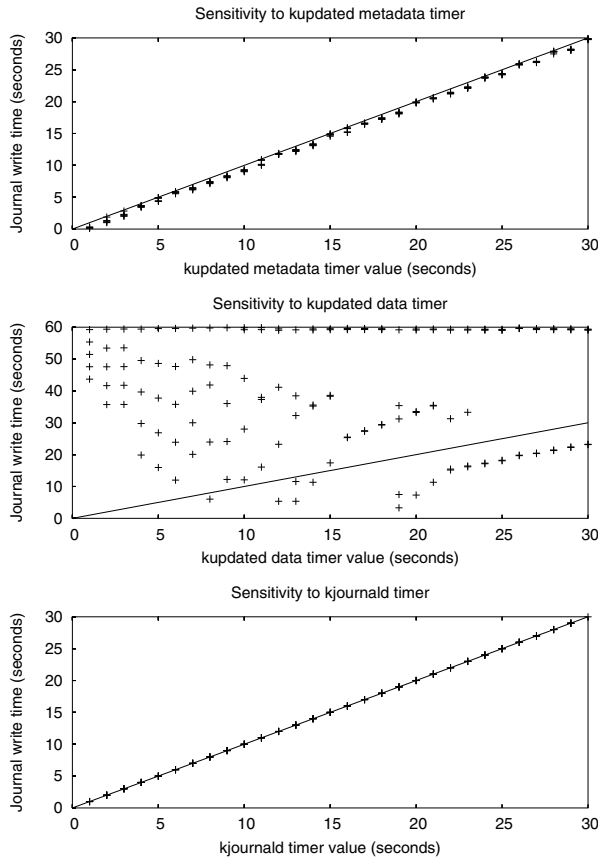


Figure 8: **Impact of Timers on Commit Policy in ext3.** In each graph, the value of one timer is varied across the x-axis, and the time of the first write to the journal is recorded along the y-axis. When measuring the impact of a particular timer, we set the other timers to 60 seconds and the journal size to 50 MB so that they do not affect the measurements.

and 30 seconds for the kupdate data timer.

We measure how these timers affect when transactions are committed to the journal. To ensure that a specific timer influences journal commits, we set the journal size to be sufficiently large and set the other timers to a large value (*i.e.*, 60 s). For our analysis, we observe when the first write appears in the journal. Figure 8 plots our results varying one of the timers along the x-axis, and plotting the time that the first log write occurs along the y-axis.

The first graph and the third graph show that the kupdate daemon metadata commit timer and the kjournal daemon commit timer control the timing of log writes: the data points along $y = x$ indicate that the log write occurred precisely when the timer expired. Thus, traffic is sent to the log at the minimum of those two timers. The second graph shows that the kupdate daemon data timer does not influence the timing of log writes: the data points are not correlated with the x-axis. As we will see, this timer influences when data is written to its fixed location.

Interaction of Journal and Fixed-Location Traffic:

The timing between writes to the journal and to the fixed-

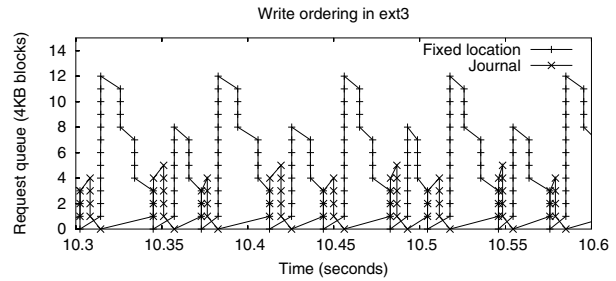


Figure 9: **Interaction of Journal and Fixed-Location Traffic in ext3.** The figure plots the number of outstanding writes to the journal and fixed-location disks. In this experiment, we run five processes, each of which issues 16 KB random synchronous writes. The file system has a 50 MB journal and is running in ordered mode; the journal is configured to run on a separate disk.

location data must be managed carefully for consistency. In fact, the difference between writeback and ordered mode is in this timing: writeback mode does not enforce any ordering between the two, whereas ordered mode ensures that the data is written to its fixed location before the commit block for that transaction is written to the journal. When we performed our SBA analysis, we found a performance deficiency in how ordered mode is implemented.

We consider a workload that synchronously writes a large number of random 16 KB blocks and use the SBA driver to separate journal and fixed-location data. Figure 9 plots the number of concurrent writes to each data type over time. The figure shows that writes to the journal and fixed-place data do *not* overlap. Specifically, ext3 issues the data writes to the fixed location and waits for completion, then issues the journal writes to the journal and again waits for completion, and finally issues the final commit block and waits for completion. We observe this behavior irrespective of whether the journal is on a separate device or on the same device as the file system. Inspection of the ext3 code confirms this observation. However, the first wait is not needed for correctness. In those cases where the journal is configured on a separate device, this extra wait can severely limit concurrency and performance. Thus, ext3 has *falsely limited parallelism*. We will use STP to fix this timing problem in §3.3.4.

3.2.3 Checkpoint Policy

We next turn our attention to checkpointing, the process of writing data to its fixed location within the ext2 structures. We will show that checkpointing in ext3 is again a function of the journal size and the commit timers, as well as the synchronization interval in the workload. We focus on data journaling mode since it is the most sensitive to journal size. To understand when checkpointing occurs, we construct workloads that periodically force data to the journal (*i.e.*, call `fsync`) and we observe when data is subsequently written to its fixed location.

Impact of Journal Size: Figure 10 shows our SBA results

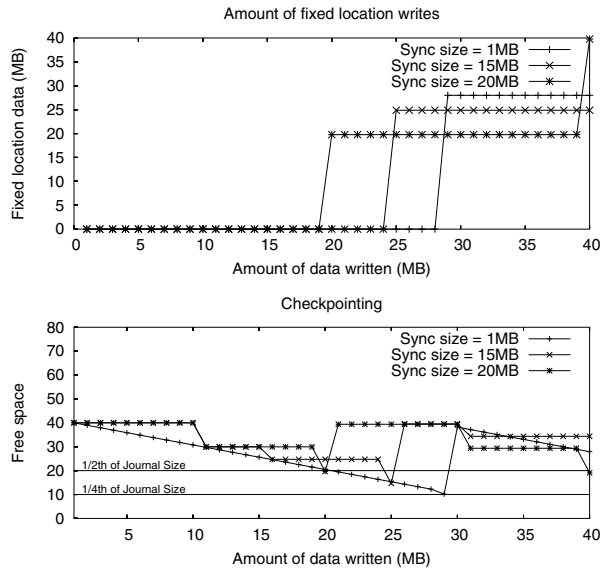


Figure 10: **Impact of Journal Size on Checkpoint Policy in ext3.** We consider a workload where a certain amount of data (as indicated by the x-axis value) is written sequentially, with a `fsync` issued after every 1, 15, or 20 MB. The first graph uses SBA to plot the amount of fixed-location traffic. The second graph uses SBA to plot the amount of free space in the journal.

as a function of file size and synchronization interval for a single journal size of 40 MB. The first graph shows the amount of data written to its fixed ext2 location at the end of each experiment. We can see that the point at which checkpointing occurs varies across the three sync intervals; for example, with a 1 MB sync interval (*i.e.*, when data is forced to disk after every 1 MB worth of writes), checkpoints occur after approximately 28 MB has been committed to the log, whereas with a 20 MB sync interval, checkpoints occur after 20 MB. To illustrate what triggers a checkpoint, in the second graph, we plot the amount of journal free space immediately preceding the checkpoint. By correlating the two graphs, we see that checkpointing occurs when the amount of free space is between $\frac{1}{4}$ -th and $\frac{1}{2}$ -th of the journal size. The precise fraction depends upon the synchronization interval, where smaller sync amounts allow checkpointing to be postponed until there is less free space in the journal.¹ We have confirmed this same relationship for other journal sizes (not shown).

Impact of Timers: We examine how the system timers impact the timing of checkpoint writes to the fixed loca-

¹The exact amount of free space that triggers a checkpoint is not straightforward to derive for two reasons. First, ext3 reserves some amount of journal space for overhead such as descriptor and commit blocks. Second, ext3 reserves space in the journal for the currently committing transaction (*i.e.*, the synchronization interval). Although we have derived the free space function more precisely, we do not feel this very detailed information is particularly enlightening; therefore, we simply say that checkpointing occurs when free space is somewhere between $\frac{1}{4}$ -th and $\frac{1}{2}$ -th of the journal size.

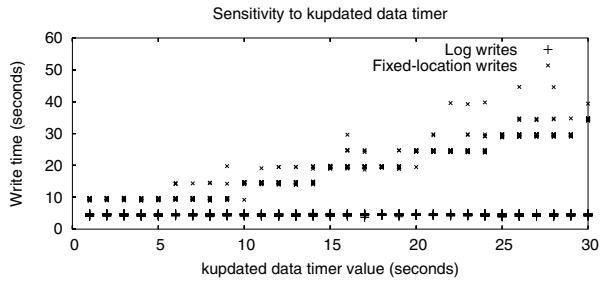


Figure 11: **Impact of Timers on Checkpoint Policy in ext3.** The figure plots the relationship between the time that data is first written to the log and then checkpointed as dependent on the value of the kupdate data timer. The scatter plot shows the results of multiple (30) runs. The process that is running writes 1 MB of data (no `fsync`); data journaling mode is used, with other timers set to 5 seconds and a journal size of 50 MB.

tions using the same workload as above. Here, we vary the kupdate data timer while setting the other timers to five seconds. Figure 11 shows how the kupdate data timer impacts when data is written to its fixed location. First, as seen previously in Figure 8, the log is updated after the five second timers expire. Then, the checkpoint write occurs later by the amount specified by the kupdate data timer, at a five second granularity; further experiments (not shown here) reveal that this granularity is controlled by the kupdate metadata timer.

Our analysis reveals that the ext3 timers do not lead to the same timing of data and metadata traffic as in ext2. Ordered and data journaling modes force data to disk either before or at the time of metadata writes. Thus, *both* data and metadata are flushed to disk frequently. This timing behavior is the largest potential performance differentiator between ordered and writeback modes. Interestingly, this frequent flushing has a potential advantage; by forcing data to disk in a more timely manner, large disk queues can be avoided and overall performance improved [18]. The disadvantage of early flushing, however, is that temporary files may be written to disk before subsequent deletion, increasing the overall load on the I/O system.

3.2.4 Summary of Ext3

Using SBA, we have isolated a number of features within ext3 that can have a strong impact on performance.

- The journaling mode that delivers the best performance depends strongly on the workload. It is well known that random workloads perform better with logging [25]; however, the relationship between the size of the journal and the amount of data written by the application can have an even larger impact on performance.

- Ext3 implements compound transactions in which unrelated concurrent updates are placed into the same transaction. The result of this *tangled synchrony* is that all traffic in a transaction is committed to disk at the same rate, which results in disastrous performance for asynchronous traffic when combined with synchronous traffic.

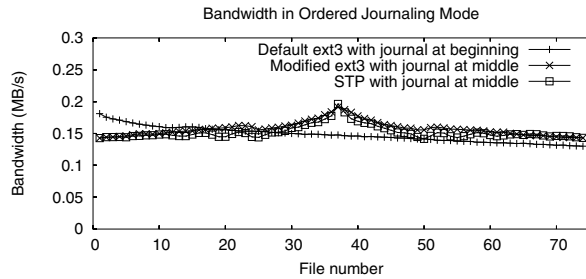


Figure 12: **Improved Journal Placement with STP.** We compare three placements of the journal: at the beginning of the partition (the ext3 default), modeled in the middle of the file system using STP, and in the middle of the file system. 50 MB files are created across the file system; a file is chosen, as indicated by the number along the x-axis, and the workload issues 4 KB synchronous writes to that file.

- In ordered mode, ext3 does not overlap any of the writes to the journal and fixed-pace data. Specifically, ext3 issues the data writes to the fixed location and waits for completion, then issues the journal writes to the journal and again waits for completion, and finally issues the final commit block and waits for completion; however, the first wait is not needed for correctness. When the journal is placed on a separate device, this *falsely limited parallelism* can harm performance.

- In ordered and data journaling modes, when a timer flushes meta-data to disk, the corresponding data must be flushed as well. The disadvantage of this *eager writing* is that temporary files may be written to disk, increasing the I/O load.

3.3 Evolving ext3 with STP

In this section, we apply STP and use a wider range of workloads and traces to evaluate various modifications to ext3. To demonstrate the accuracy of the STP approach, we begin with a simple modification that varies the placement of the journal. Our SBA analysis pointed to a number of improvements for ext3, which we can quantify with STP: the value of using different journaling modes depending upon the workload, having separate transactions for each update, and overlapping pre-commit journal writes with data updates in ordered mode. Finally, we use STP to evaluate *differential journaling*, in which block differences are written to the journal.

3.3.1 Journal Location

Our first experiment with STP quantifies the impact of changing a simple policy: the placement of the journal. The default ext3 creates the journal as a regular file at the beginning of the partition. We start with this policy because we are able to validate STP: the results we obtain with STP are quite similar to those when we implement the change within ext3 itself.

We construct a workload that stresses the placement of the journal: a 4 GB partition is filled with 50 MB files and the benchmark process issues random, synchronous

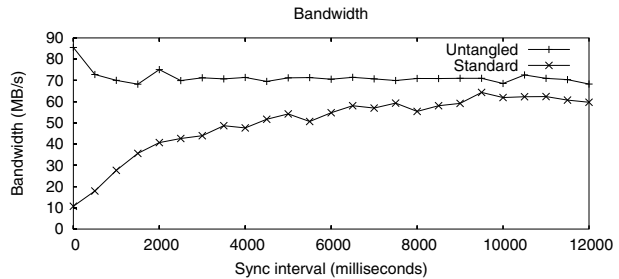


Figure 13: **Untangling Transaction Groups with STP.** This experiment is identical to that described in Figure 6, with one addition: we show performance of the foreground process with untangled transactions as emulated with STP.

4 KB writes to a chosen file. In Figure 12 we vary which file is chosen along the x-axis. The first line in the graph shows the performance for ordered mode in default ext3: bandwidth drops by nearly 30% when the file is located far from the journal. SBA analysis (not shown) confirms that this performance drop occurs as the seek distance increases between the writes to the file and the journal.

To evaluate the benefit of placing the journal in the middle of the disk, we use STP to remap blocks. For validation, we also coerce ext3 to allocate its journal in the middle of the disk, and compare results. Figure 12 shows that the STP predicted performance is nearly identical to this version of ext3. Furthermore, we see that worst-case behavior is avoided; by placing the journal in the middle of the file system instead of at the beginning, the longest seeks across the entire volume are avoided during synchronous workloads (*i.e.*, workloads that frequently seek between the journal and the ext2 structures).

3.3.2 Journaling Mode

As shown in §3.2.1, different workloads perform better with different journaling modes. For example, random writes perform better in data journaling mode as the random writes are written sequentially into the journal, but large sequential writes perform better in ordered mode as it avoids the extra traffic generated by data journaling mode. However, the journaling mode in ext3 is set at mount time and remains fixed until the next mount.

Using STP, we evaluate a new adaptive journaling mode that chooses the journaling mode for each transaction according to writes that are in the transaction. If a transaction is sequential, it uses ordered journaling; otherwise, it uses data journaling.

To demonstrate the potential performance benefits of adaptive journaling, we run a portion of a trace from HP Labs [23] after removing the inter-arrival times between the I/O calls and compare ordered mode, data journaling mode, and our adaptive approach. The trace completes in 83.39 seconds and 86.67 seconds, in ordered and data journaling modes, respectively; however, with STP adaptive journaling, the trace completes in only 51.75 seconds. Because the trace has both sequential and random write

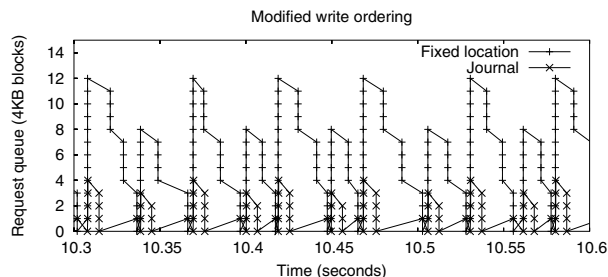


Figure 14: **Changing the Interaction of Journal and Fixed-Location Traffic with STP.** The same experiment is run as in Figure 9; however, in this run, we use STP to issue the pre-commit journal writes and data writes concurrently. We plot the STP emulated performance, and also made this change to ext3 directly, obtaining the same resultant performance.

phases, adaptive journaling outperforms any single-mode approach.

3.3.3 Transaction Grouping

Linux ext3 groups all updates into system-wide compound transactions and commits them to disk periodically. However, as we have shown in 3.2.1, if just a single update stream is synchronous, it can have a dramatic impact on the performance of other asynchronous streams, by transforming in-memory updates into disk-bound ones.

Using STP, we show the performance of a file system that *untangles* these traffic streams, only forcing the process that issues the `fsync` to commit its data to disk. Figure 13 plots the performance of an asynchronous sequential stream in the presence of a random synchronous stream. Once again, we vary the interval of updates from the synchronous process, and from the graph, we can see that segregated transaction grouping is effective; the asynchronous I/O stream is unaffected by synchronous traffic.

3.3.4 Timing

We show that STP can quantify the cost of falsely limited parallelism, as discovered in 3.2.2, where pre-commit journal writes are not overlapped with data updates in ordered mode. With STP, we modify the timing so that journal and fixed-location writes are all initiated simultaneously; the commit transaction is written only after the previous writes complete. We consider the same workload of five processes issuing 16 KB random synchronous writes and with the journal on a separate disk.

Figure 14 shows that STP can model this implementation change by modifying the timing of the requests. For this workload, STP predicts an improvement of about 18%; this prediction matches what we achieve when ext3 is changed directly. Thus, as expected, increasing the amount of concurrency improves performance when the journal is on a separate device.

3.3.5 Journal Contents

Ext3 uses physical logging and writes new blocks in their entirety to the log. However, if whole blocks are jour-

naled irrespective of how many bytes have changed in the block, journal space fills quickly, increasing both commit and checkpoint frequency.

Using STP, we investigate *differential journaling*, where the file system writes block differences to the journal instead of new blocks in their entirety. This approach can potentially reduce disk traffic noticeably, if dirty blocks are not substantially different from their previous versions. We focus on data journaling mode, as it generates by far the most journal traffic; differential journaling is less useful for the other modes.

To evaluate whether differential journaling matters for real workloads, we analyze SBA traces underneath two database workloads modeled on TPC-B [30] and TPC-C [31]. The former is a simple application-level implementation of a debit-credit benchmark, and the latter a realistic implementation of order-entry built on top of Postgres. With data journaling mode, the amount of data written to the journal is reduced by a factor of 200 for TPC-B and a factor of 6 under TPC-C. In contrast, for ordered and writeback modes, the difference is minimal (less than 1%); in these modes, only metadata is written to the log, and applying differential journaling to said metadata blocks makes little difference in total I/O volume.

4 ReiserFS

We now focus on a second Linux journaling filesystem, ReiserFS. In this section, we focus on the chief differences between ext3 and ReiserFS. Due to time constraints, we do not use STP to explore changes to ReiserFS.

4.1 Background

The general behavior of ReiserFS is similar to ext3. For example, both file systems have the same three journaling modes and both have compound transactions. However, ReiserFS differs from ext3 in three primary ways.

First, the two file systems use different on-disk structures to track their fixed-location data. Ext3 uses the same structures as ext2; for improved scalability, ReiserFS uses a B+ tree, in which data is stored on the leaves of the tree and the metadata is stored on the internal nodes. Since the impact of the fixed-location data structures is not the focus of this paper, this difference is largely irrelevant.

Second, the format of the journal is slightly different. In ext3, the journal can be a file, which may be anywhere in the partition and may not be contiguous. The ReiserFS journal is not a file and is instead a contiguous sequence of blocks at the beginning of the file system; as in ext3, the ReiserFS journal can be put on a different device. Further, ReiserFS limits the journal to a maximum of 32 MB.

Third, ext3 and ReiserFS differ slightly in their journal contents. In ReiserFS, the fixed locations for the blocks in the transaction are stored not only in the descriptor block but also in the commit block. Also, unlike ext3, ReiserFS uses only one descriptor block in every compound

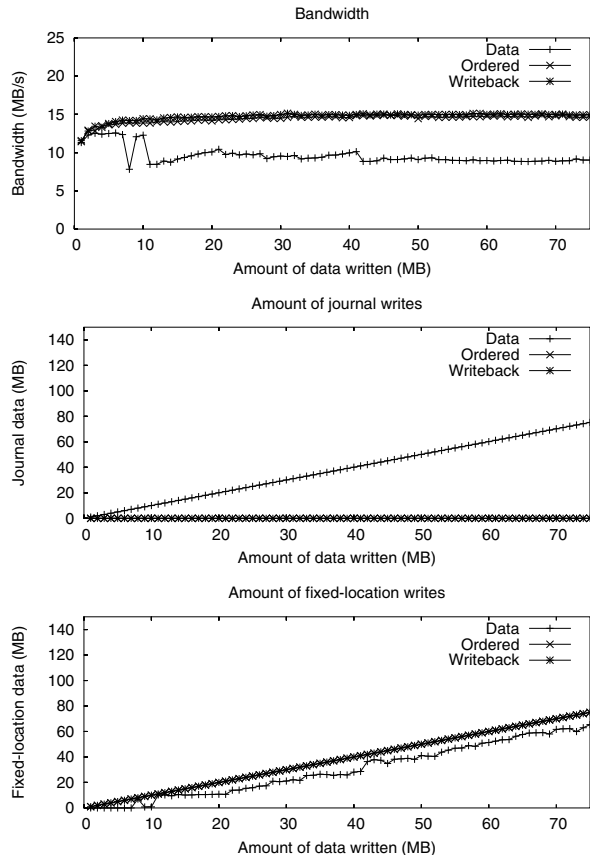


Figure 15: **Basic Behavior for Sequential Workloads in ReiserFS.** Within each graph, we evaluate the three ReiserFS journaling modes. We consider a single workload in which the size of the sequentially written file is increased along the x-axis. Each graph examines a different metric: the first shows the achieved bandwidth; the second uses SBA to report the amount of journal traffic; the third uses SBA to report the amount of fixed-location traffic. The journal size is set to 32 MB.

transaction, which limits the number of blocks that can be grouped in a transaction.

4.2 Semantic Analysis of ReiserFS

We have performed identical experiments on ReiserFS as we have on ext3. Due to space constraints, we present only those results which reveal significantly different behavior across the two file systems.

4.2.1 Basic Behavior: Modes and Workload

Qualitatively, the performance of the three journaling modes in ReiserFS is similar to that of ext3: random workloads with infrequent synchronization perform best with data journaling; otherwise, sequential workloads generally perform better than random ones and write-back and ordered modes generally perform better than data journaling. Furthermore, ReiserFS groups concurrent transactions into a single compound transaction, as did ext3. The primary difference between the two file systems occurs for sequential workloads with data journaling. As shown in the first graph of Figure 15, the

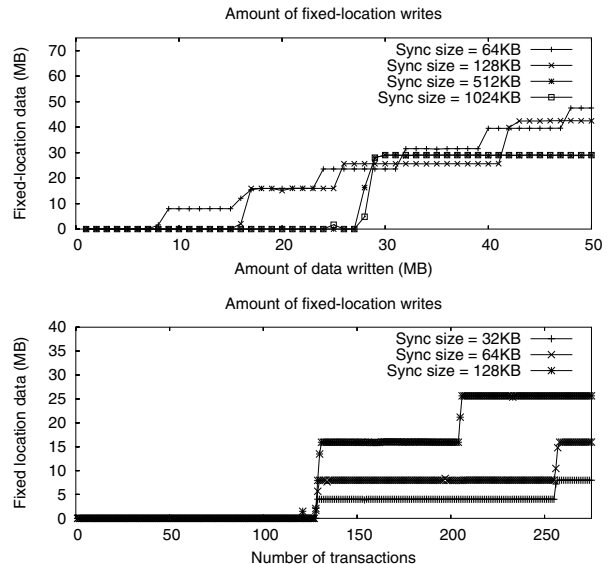


Figure 16: **Impact of Journal Size and Transactions on Checkpoint Policy in ReiserFS.** We consider workloads where data is sequentially written and an `fsync` is issued after a specified amount of data. We use SBA to report the amount of fixed-location traffic. In the first graph, we vary the amount of data written; in the second graph, we vary the number of transactions, defined as the number of calls to `fsync`.

throughput of data journaling mode in ReiserFS does not follow the sawtooth pattern. An initial reason for this is found through SBA analysis. As seen in the second and third graphs of Figure 15, almost all of the data is written not only to the journal, but is also checkpointed to its in-place location. Thus, ReiserFS appears to checkpoint data much more aggressively than ext3, which we will explore in §4.2.3.

4.2.2 Journal Commit Policy

We explore the factors that impact when ReiserFS commits transactions to the log. Again, we focus on data journaling, since it is the most sensitive. We postpone exploring the impact of the timers until §4.2.3.

We previously saw that ext3 commits data to the log when approximately $\frac{1}{4}$ of the log is filled or when a timer expires. Running the same workload that does not force data to disk (*i.e.*, does not call `fsync`) on ReiserFS and performing SBA analysis, we find that ReiserFS uses a different threshold: depending upon whether the journal size is below or above 8 MB, ReiserFS commits data when about 450 blocks (*i.e.*, 1.7 MB) or 900 blocks (*i.e.*, 3.6 MB) are written. Given that ReiserFS limits journal size to at most 32 MB, these fixed thresholds appear sufficient.

Finally, we note that ReiserFS also has falsely limited parallelism in ordered mode. Like ext3, ReiserFS forces the data to be flushed to its fixed location before it issues any writes to the journal.

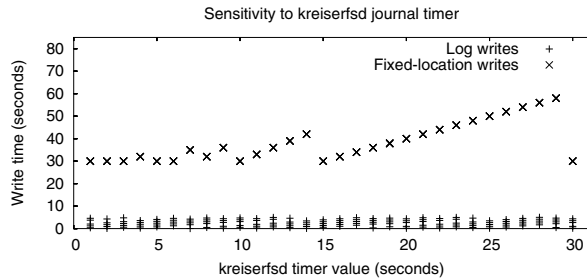


Figure 17: **Impact of Timers in ReiserFS.** The figure plots the relationship between the time that data is written and the value of the *kreiserfs* timer. The scatter plot shows the results of multiple (30) runs. The process that is running writes 1 MB of data (no *fsync*); data journaling mode is used, with other timers set to 5 seconds and a journal size of 32 MB.

4.2.3 Checkpoint Policy

We also investigate the conditions which trigger ReiserFS to checkpoint data to its fixed-place location; this policy is more complex in ReiserFS. In *ext3*, we found that data was checkpointed when the journal was $\frac{1}{4}$ to $\frac{1}{2}$ full. In ReiserFS, the point at which data is checkpointed depends not only on the free space in the journal, but also on the number of concurrent transactions. We again consider workloads that periodically force data to the journal by calling *fsync* at different intervals.

Our results are shown in Figure 16. The first graph shows the amount of data checkpointed as a function of the amount of data written; in all cases, data is checkpointed before $\frac{7}{8}$ of the journal is filled. The second graph shows the amount of data checkpointed as a function of the number of transactions. This graph shows that data is checkpointed at least at intervals of 128 transactions; running a similar workload on *ext3* reveals no relationship between the number of transactions and checkpointing. Thus, ReiserFS checkpoints data whenever either journal free space drops below 4 MB or when there are 128 transactions in the journal.

As with *ext3*, timers control when data is written to the journal and to the fixed locations, but with some differences: in *ext3*, the *kjournal* daemon is responsible for committing transactions, whereas in ReiserFS, the *kreiserfs* daemon has this role. Figure 17 shows the time at which data is written to the journal and to the fixed location as the *kreiserfs* timer is increased; we make two conclusions. First, log writes always occur within the first five seconds of the data write by the application, regardless of the timer value. Second, the fixed-location writes occur only when the elapsed time is both greater than 30 seconds and a multiple of the *kreiserfs* timer value. Thus, the ReiserFS timer policy is simpler than that of *ext3*.

4.3 Finding Bugs

SBA analysis is useful not only for inferring the policies of filesystems, but also for finding cases that have not been implemented correctly. With SBA analysis, we

have found a number of problems with the ReiserFS implementation that have not been reported elsewhere. In each case, we identified the problem because the SBA driver did not observe some disk traffic that it expected. To verify these problems, we have also examined the code to find the cause and have suggested corresponding fixes to the ReiserFS developers.

- In the first transaction after a mount, the *fsync* call returns before any of the data is written. We tracked this aberrant behavior to an incorrect initialization.
- When a file block is overwritten in writeback mode, its stat information is not updated. This error occurs due to a failure to update the inode's transaction information.
- When committing old transactions, dirty data is not always flushed. We tracked this to erroneously applying a condition to prevent data flushing during journal replay.
- Irrespective of changing the journal thread's wake up interval, dirty data is not flushed. This problem occurs due to a simple coding error.

5 The IBM Journaled File System

In this section, we describe our experience performing a preliminary SBA analysis of the Journaled File System (JFS). We began with a rudimentary understanding of JFS from what we were able to obtain through documentation [3]; for example, we knew that the journal is located by default at the end of the partition and is treated as a contiguous sequence of blocks and that one cannot specify the journaling mode.

Due to the fact that we knew less about this file system before we began, we found we needed to apply a new analysis technique as well: in some cases we filtered out traffic and then rebooted the system so that we could infer whether the filtered traffic was necessary for consistency or not. For example, we used this technique to understand the journaling mode of JFS. From this basic starting point, and without examining JFS code, we were able to learn a number of interesting properties about JFS.

First, we inferred that JFS uses ordered journaling mode. Due to the small amount of traffic to the journal, it was obvious that it was not employing data journaling. To differentiate between writeback and ordered modes, we observed that the ordering of writes matched that of ordered mode. That is, when a data block is written by the application, JFS orders the write such that the data block is written successfully before the metadata writes are issued.

Second, we determined that JFS does logging at the record level. That is, whenever an inode, index tree, or directory tree structure changes, only that structure is logged instead of the entire block containing the structure. As a result, JFS writes fewer journal blocks than *ext3* and ReiserFS for the same operations.

Third, JFS does not by default group concurrent updates into a single compound transaction. Running the same experiment as we performed in Figure 6, we see that

the bandwidth of the asynchronous traffic is very high irrespective of whether there is a synchronous traffic in the background. However, there are circumstances in which transactions are grouped: for example, if the write commit records are on the same log page.

Finally, there are no commit timers in JFS and the fixed-location writes happen whenever the kupdate daemon's timer expires. However, the journal writes are never triggered by the timer: journal writes are indefinitely postponed until there is another trigger such as memory pressure or an unmount operation. This *infinite write delay* limits reliability, as a crash can result in data loss even for data that was written minutes or hours before.

6 Windows NTFS

In this section, we explain our analysis of NTFS. NTFS is a journaling file system that is used as the default file system on Windows operating systems such as XP, 2000, and NT. Although the source code or documentation of NTFS is not publicly available, tools for finding the NTFS file layout exist [28].

We ran the Windows XP operating system on top of VMware on a Linux machine. The pseudo device driver was exported as a SCSI disk to the Windows and a NTFS file system was constructed on top of the pseudo device. We ran simple workloads on NTFS and observed traffic within the SBA driver for our analysis.

Every object in NTFS is a file. Even metadata is stored in terms of files. The journal itself is a file and is located almost at the center of the file system. We used the ntfsprogs tools to discover journal file boundaries. Using the journal boundaries we were able to distinguish journal traffic from fixed-location traffic.

From our analysis, we found that NTFS does not do data journaling. This can be easily verified by the amount of data traffic observed by the SBA driver. We also found that NTFS, similar to JFS, does not do block-level journaling. It journals metadata in terms of records. We verified that whole blocks are not journaled in NTFS by matching the contents of the fixed-location traffic to the contents of the journal traffic.

We also inferred that NTFS performs ordered journaling. On data writes, NTFS waits until the data block writes to the fixed-location complete before writing the metadata blocks to the journal. We confirmed this ordering by using the SBA driver to delay the data block writes upto 10 seconds and found that the following metadata writes to the journal are delayed by the corresponding amount.

7 Related Work

Journaling Studies: Journaling file systems have been studied in detail. Most notably, Seltzer *et al.* [26] compare two variants of a journaling FFS to soft updates [11], a different technique for managing metadata consistency for file systems. Although the authors present no direct

observation of low-level traffic, they are familiar enough with the systems (indeed, they are the implementors!) to explain behavior and make “semantic” inferences. For example, to explain why journaling performance drops in a delete benchmark, the authors report that the file system is “forced to read the first indirect block in order to reclaim the disk blocks it references” ([26], Section 8.1). A tool such as SBA makes such expert observations more readily available to all. Another recent study compares a range of Linux file systems, including ext2, ext3, ReiserFS, XFS, and JFS [7]. This work evaluates which file systems are fastest for different benchmarks, but gives little explanation as to *why* one does well for a given workload.

File System Benchmarks: There are many popular file system benchmarks, such as IOzone [19], Bonnie [6], Imbench [17], the modified Andrew benchmark [20], and PostMark [14]. Some of these (IOZone, Bonnie, Imbench) perform synthetic read/write tests to determine throughput; others (Andrew, Postmark) are intended to model “realistic” application workloads. Uniformly, all measure overall throughput or runtime to draw high-level conclusions about the file system. In contrast to SBA, none are intended to yield low-level insights about the internal policies of the file system.

Perhaps the most related to our work is Chen and Patterson's self-scaling benchmark [8]. In this work, the benchmarking framework conducts a search over the space of possible workload parameters (*e.g.*, sequentiality, request size, total workload size, and concurrency), and hones in on interesting parts of the workload space. Interestingly, some conclusions about file system behavior can be drawn from the resultant output, such as the size of the file cache. Our approach is not nearly as automated; instead, we construct benchmarks that exercise certain file system behaviors in a controlled manner.

File System Tracing: Many previous studies have traced file system activity. For example, Zhou *et al.* [37], Ousterhout *et al.* [21], Baker *et al.* [2], and Roselli *et al.* [24] all record various file system operations to later deduce file-level access patterns. Vogels [35] performs a similar study but inside the NT file system driver framework, where more information is available (*e.g.*, mapped I/O is not missed, as it is in most other studies). A recent example of a tracing infrastructure is TraceFS [1], which traces file systems at the VFS layer; however, TraceFS does not enable the low-level tracing that SBA provides. Finally, Blaze [5] and later Ellard *et al.* [10] show how low-level packet tracing can be useful in an NFS environment. By recording network-level protocol activity, network file system behavior can be carefully analyzed. This type of packet analysis is analogous to SBA since they are both positioned at a low level and thus must reconstruct higher-level behaviors to obtain a complete view.

8 Conclusions

As systems grow in complexity, there is a need for techniques and approaches that enable both users and system architects to understand in detail how such systems operate. We have presented semantic block-level analysis (SBA), a new methodology for file system benchmarking that uses block-level tracing to provide insight about the internal behavior of a file system. The block stream annotated with semantic information (*e.g.*, whether a block belongs to the journal or to another data structure) is an excellent source of information.

In this paper, we have focused on how the behavior of journaling file systems can be understood with SBA. In this case, using SBA is very straightforward: the user must know only how the journal is allocated on disk. Using SBA, we have analyzed in detail two Linux journaling file systems: ext3 and ReiserFS. We also have performed a preliminary analysis of Linux JFS and Windows NTFS. In all cases, we have uncovered behaviors that would be difficult to discover using more conventional approaches.

We have also developed and presented semantic trace playback (STP) which enables the rapid evaluation of new ideas for file systems. Using STP, we have demonstrated the potential benefits of numerous modifications to the current ext3 implementation for real workloads and traces. Of these modifications, we believe the transaction grouping mechanism within ext3 should most seriously be reevaluated; an untangled approach enables asynchronous processes to obtain in-memory bandwidth, despite the presence of other synchronous I/O streams in the system.

Acknowledgments

We thank Theodore Ts'o, Jiri Schindler and the members of the ADSL research group for their insightful comments. We also thank Mustafa Uysal for his excellent shepherding, and the anonymous reviewers for their thoughtful suggestions. This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM and EMC.

References

- [1] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *FAST '04*, San Francisco, CA, April 2004.
- [2] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *SOSP '91*, pages 198–212, Pacific Grove, CA, October 1991.
- [3] S. Best. JFS Log. How the Journaled File System performs logging. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 163–168, Atlanta, 2000.
- [4] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2004.
- [5] M. Blaze. NFS tracing by passive network monitoring. In *USENIX Winter '92*, pages 333–344, San Francisco, CA, January 1992.
- [6] T. Bray. The Bonnie File System Benchmark. <http://www.textuality.com/bonnie/>.
- [7] R. Bryant, R. Forester, and J. Hawkes. Filesystem Performance and Scalability in Linux 2.4.17. In *FREENIX '02*, Monterey, CA, June 2002.
- [8] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation—Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *SIGMETRICS '93*, pages 1–12, Santa Clara, CA, May 1993.
- [9] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode File System. In *USENIX Winter '92*, pages 43–60, San Francisco, CA, January 1992.
- [10] D. Ellard and M. I. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *LISA '03*, pages 73–85, San Diego, California, October 2003.
- [11] G. R. Ganger and Y. N. Patt. Metadata Update Performance in File Systems. In *OSDI '94*, pages 49–60, Monterey, CA, November 1994.
- [12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [13] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87*, Austin, Texas, November 1987.
- [14] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.
- [15] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.
- [16] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsync - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.
- [17] L. McVoy and C. Staelin. Imbench: Portable Tools for Performance Analysis. In *USENIX 1996*, San Diego, CA, January 1996.
- [18] J. C. Mogul. A Better Update Policy. In *USENIX Summer '94*, Boston, MA, June 1994.
- [19] W. Norcutt. The IOzone Filesystem Benchmark. <http://www.iozone.org/>.
- [20] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the 1990 USENIX Summer Technical Conference*, Anaheim, CA, June 1990.
- [21] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *SOSP '85*, pages 15–24, Orcas Island, WA, December 1985.
- [22] H. Reiser. ReiserFS. www.namesys.com, 2004.
- [23] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *FAST '02*, pages 14–29, Monterey, CA, January 2002.
- [24] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *USENIX '00*, pages 41–54, San Diego, California, June 2000.
- [25] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
- [26] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *USENIX '00*, pages 71–84, San Diego, California, June 2000.
- [27] D. A. Solomon. *Inside Windows NT (Microsoft Programming Series)*. Microsoft Press, 1998.
- [28] SourceForge. The Linux NTFS Project. <http://linux-ntfs.sf.net/>, 2004.
- [29] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *USENIX 1996*, San Diego, CA, January 1996.
- [30] Transaction Processing Council. TPC Benchmark B Standard Specification, Revision 3.2. Technical Report, 1990.
- [31] Transaction Processing Council. TPC Benchmark C Standard Specification, Revision 5.2. Technical Report, 1992.
- [32] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *FREENIX '02*, Monterey, CA, June 2002.
- [33] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.
- [34] S. C. Tweedie. EXT3, Journaling File System. olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html, July 2000.
- [35] W. Vogels. File system usage in Windows NT 4.0. In *SOSP '99*, pages 93–109, Kiawah Island Resort, SC, December 1999.
- [36] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, San Francisco, CA, December 2004.
- [37] S. Zhou, H. D. Costa, and A. Smith. A File System Tracing Package for Berkeley UNIX. In *USENIX Summer '84*, pages 407–419, Salt Lake City, UT, June 1984.

Comparison-based File Server Verification

Yuen-Lin Tan*, Terrence Wong, John D. Strunk, Gregory R. Ganger
Carnegie Mellon University

Abstract

Comparison-based server verification involves testing a server by comparing its responses to those of a reference server. An intermediary, called a “server Tee,” interposes between clients and the reference server, synchronizes the system-under-test (SUT) to match the reference server’s state, duplicates each request for the SUT, and compares each pair of responses to identify any discrepancies. The result is a detailed view into any differences in how the SUT satisfies the client-server protocol specification, which can be invaluable in debugging servers, achieving bug compatibility, and isolating performance differences. This paper introduces, develops, and illustrates the use of comparison-based server verification. As a concrete example, it describes a NFSv3 Tee and reports on its use in identifying interesting differences in several production NFS servers and in debugging a prototype NFS server. These experiences confirm that comparison-based server verification can be a useful tool for server implementors.

1 Introduction

Debugging servers is tough. Although the client-server interface is usually documented in a specification, there are often vague or unspecified aspects. Isolating specification interpretation flaws in request processing and in responses can be a painful activity. Worse, a server that works with one type of client may not work with another, and testing with all possible clients is not easy.

The most common testing practices are RPC-level test suites and benchmarking with one or more clients. With enough effort, one can construct a suite of tests that exercises each RPC in a variety of cases and verifies that each response conforms to what the specification dictates. This is a very useful approach, though time-consuming to develop and difficult to perfect in the face of specification vagueness. Popular benchmark programs, such as SPEC SFS [15] for NFS servers, are often used to stress-test servers and verify that they work for the clients used in the benchmark runs.

This paper proposes an additional tool for server testing: *comparison-based server verification*. The idea is sim-

*Currently works for VMware.

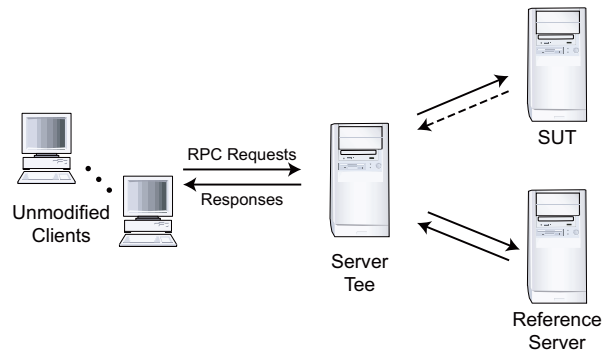


Figure 1: Using a server Tee for comparison-based verification. The server Tee is interposed between unmodified clients and the unmodified reference server, relaying requests and responses between them. The Tee also sends the same requests to the system-under-test and compares the responses to those from the reference server. With the exception of performance interference, this latter activity should be invisible to the clients.

ple: each request is sent to both the system-under-test (SUT) and a *reference server*, and the two responses are compared. This can even be done in a live environment with real clients to produce scenarios that artificial test suites may miss. The reference server is chosen based on the belief that it is a valid implementation of the relevant interface specification. For example, it might be a server that has been used for some time by many user communities. The reference server thus becomes a “gold standard” against which the SUT’s conformity can be evaluated. Given a good reference server, comparison-based server verification can assist with debugging infrequent problems, achieving “bug compatibility,” and isolating performance differences.

This paper specifically develops the concept of comparison-based verification of file servers via use of a *file server Tee* (See Figure 1).¹ A file server Tee interposes on communication between clients and the reference server. The Tee automatically sets and maintains SUT state (i.e., directories, files, etc.) to match the reference server’s state, forwards client requests to the reference server, duplicates client requests for the SUT, and compares the two responses for each request. Only the reference server’s responses are sent to clients, which

¹The name, “server Tee,” was inspired by the UNIX `tee` command, which reads data from standard input and writes it to both standard output and one or more output files.

makes it possible to perform comparison-based verification even in live environments.

The paper details the design and implementation of a NFSv3 Tee. To illustrate the use of a file server Tee, we present the results of using our NFSv3 Tee to compare several popular production NFS servers, including FreeBSD, a Network Appliance box, and two versions of Linux. A variety of differences are identified, including some discrepancies that would affect correctness for some clients. We also describe experiences using our NFSv3 Tee to debug a prototype NFS server.

The remainder of this paper is organized as follows. Section 2 puts comparison-based server verification in context and discusses what it can be used for. Section 3 discusses how a file server Tee works. Section 4 describes the design and implementation of our NFSv3 Tee. Section 5 evaluates our NFSv3 Tee and presents results of several case studies using it. Section 6 discusses additional issues and features of comparison-based file server verification. Section 7 discusses related work.

2 Background

Distributed computing based on the client-server model is commonplace. Generally speaking, this model consists of clients sending RPC requests to servers and receiving responses after the server finishes the requested action. For most file servers, for example, system calls map roughly to RPC requests, supporting actions like file creation and deletion, data reads and writes, and fetching of directory entry listings.

Developing functional servers can be fairly straightforward, given the variety of RPC packages available and the maturity of the field. Fully debugging them, however, can be tricky. While the server interface is usually codified in a specification, there are often aspects that are insufficiently formalized and thus open to interpretation. Different client or server implementors may interpret them differently, creating a variety of de facto standards to be supported (by servers or clients).

There are two common testing strategies for servers. The first, based on RPC-level test suites, exercises each individual RPC request and verifies proper responses in specific situations. For each test case, the test scaffolding sets server state as needed, sends the RPC request, and compares the response to the expected value. Verifying that the RPC request did the right thing may involve additional server state checking via follow-up RPC requests. After each test case, any residual server state is cleaned up. Constructing exhaustive RPC test suites is a painstaking task, but it is a necessary step if serious robustness is desired. One challenge with such test

suites, as with almost all testing, is balancing coverage with development effort and test completion time. Another challenge, related to specification vagueness, is accuracy: the test suite implementor interprets the specification, but may not do so the same way as others.

The second testing strategy is to experiment with applications and benchmarks executing on one or more client implementation(s).² This complements RPC-level testing by exercising the server with specific clients, ensuring that those clients work well with the server when executing at least some important workloads; thus, it helps with the accuracy issue mentioned above. On the other hand, it usually offers much less coverage than RPC-level testing. It also does not ensure that the server will work with clients that were not tested.

2.1 Comparison-based verification

Comparison-based verification complements these testing approaches. It does not eliminate the coverage problem, but it can help with the accuracy issue by conforming to someone else's interpretation of the specification. It can help with the coverage issue, somewhat, by exposing problem "types" that recur across RPCs and should be addressed en masse.

Comparison-based verification consists of comparing the server being tested to a "gold standard," a reference server whose implementation is believed to work correctly. Specifically, the state of the SUT is set up to match that of the reference server, and then each RPC request is duplicated so that the two servers' responses to each request can be compared. If the server states were synchronized properly, and the reference server is correct, differences in responses indicate potential problems with the SUT.

Comparison-based verification can help server development in four ways: debugging client-perceived problems, achieving bug compatibility with existing server implementations, testing in live environments, and isolating performance differences.

1. Debugging: With benchmark-based testing, in particular, bugs exhibit themselves as situations where the benchmark fails to complete successfully. When this happens, significant effort is often needed to determine exactly what server response(s) caused the client to fail. For example, single-stepping through client actions might be used, but this is time-consuming and may alter client behavior enough that the problem no longer arises. Another approach is to sniff network packets and interpret the exchanges between client and server to identify the last interactions before problems arise. Then, one

²Research prototypes are almost always tested only in this way.

can begin detailed analysis of those RPC requests and responses.

Comparison-based verification offers a simpler solution, assuming that the benchmark runs properly when using the reference server. Comparing the SUT's responses to the problem-free responses produced by the reference server can quickly identify the specific RPC requests for which there are differences. Comparison provides the most benefit when problems involve nuances in responses that cause problems for clients (as contrasted with problems where the server crashes)—often, these will be places where the server implementors interpreted the specification differently. For such problems, the exact differences between the two servers' responses can be identified, providing detailed guidance to the developer who needs to find and fix the implementation problem.

2. Bug compatibility: In discussing vagueness in specifications, we have noted that some aspects are often open to interpretation. Sometimes, implementors misinterpret them even if they are not vague. Although it is tempting to declare both situations “the other implementor’s problem,” that is simply not a viable option for those seeking to achieve widespread use of their server. For example, companies attempting to introduce a new server product into an existing market **must** make that server work for the popular clients. Thus, deployed clients introduce de facto standards that a server must accommodate. Further, if clients (existing and new) conform to particular “features” of a popular server’s implementation (or a previous version of the new server), then that again becomes a de facto standard. Some use the phrase, “bug compatibility,” to describe what must be achieved given these issues.

As a concrete example of bug compatibility, consider the following real problem encountered with a previous NFSv2 server we developed: Linux clients (at the time) did not invalidate directory cookies when manipulating directories, which our interpretation of the specification (and the implementations of some other clients) indicated should be done. So, with that Linux client, an “rm -rf” of a large directory would read part of the directory, remove those files, and then do another READDIR with the cookie returned by the first READDIR. Our server compressed directories when entries were removed, and thus the old cookie (an index into the directory) would point beyond some live entries after some files were removed—the “rm -rf” would thus miss some files. We considered keeping a table of cookie-to-index mappings instead, but without a way to invalidate entries safely (there are no definable client sessions in NFSv2), the table would have to be kept persistently; we finally just disabled directory compression. (NFSv3 has a “cookie verifier,” which would allow a server to solve

this problem, even when other clients change the directory.)

Comparison-based verification is a great tool for achieving bug compatibility. Specifically, one can compare each response from the SUT with that produced by a reference server that implements the de facto standard. Such comparisons expose differences that might indicate differing interpretations of the specification or other forms of failure to achieve bug compatibility. Of course, one needs an input workload that has good coverage to fully uncover de facto standards.

3. In situ verification: Testing and benchmarking allow offline verification that a server works as desired, which is perfect for those developing a new server. These approaches are of less value to IT administrators seeking comfort before replacing an existing server with a new one. In high-end environments (e.g., bank data centers), expensive service agreements and penalty clauses can provide the desired comfort. But, in less resource-heavy environments (e.g., university departments or small businesses), administrators often have to take the plunge with less comfort.

Comparison-based verification offers an alternative, which is to run the new server as the SUT for a period of time while using the existing server as the reference server.³ This requires inserting a server Tee into the live environment, which could introduce robustness and performance issues. But, because only the reference server’s responses are sent to clients, this approach can support reasonably safe in situ verification.

4. Isolating performance differences: Performance comparisons are usually done with benchmarking. Some benchmarks provide a collection of results on different types of server operations, while others provide overall application performance for more realistic workloads.

Comparison-based verification could be adapted to performance debugging by comparing per-request response times as well as response contents. Doing so would allow detailed request-by-request profiles of performance differences between servers, perhaps in the context of application benchmark workloads where disappointing overall performance results are observed. Such an approach might be particularly useful, when combined with in situ verification, for determining what benefits might be expected from a new server being considered.

³Although not likely to be its most popular use, this was our original reason for exploring this idea. We are developing a large-scale storage service to be deployed and maintained on the Carnegie Mellon campus as a research expedition into self-managing systems [4]. We wanted a way to test new versions in the wild before deploying them. We also wanted a way to do live experiments safely in the deployed environment, which is a form of the fourth item.

3 Components of a file system Tee

Comparison-based server verification happens at an interposition point between clients and servers. Although there are many ways to do this, we believe it will often take the form of a distinct proxy that we call a “server Tee”. This section details what a server Tee is by describing its four primary tasks. The subsequent section describes the design and implementation of a server Tee for NFSv3.

Relaying traffic to/from reference server: Because it interposes, a Tee must relay RPC requests and responses between clients and the reference server. The work involved in doing so depends on whether the Tee is a passive or an active intermediary. A passive intermediary observes the client-server exchanges but does not manipulate them at all—this minimizes the relaying effort, but increases the effort for the duplicating and comparing steps, which now must reconstruct RPC interactions from the observed packet-level communications. An active intermediary acts as the server for clients and as the only client for the server—it receives and parses the RPC requests/responses and generates like messages for the final destination. Depending on the RPC protocol, doing so may require modifying some fields (e.g., request IDs since all will come from one system, the Tee), which is extra work. The benefit is that other Tee tasks are simplified.

Whether a Tee is an active intermediary or a passive one, it must see all accesses that affect server state in order to avoid flagging false positives. For example, an unseen file write to the reference server would cause a subsequent read to produce a mismatch during comparison that has nothing to do with the correctness of the SUT. One consequence of the need for complete interposing is that tapping the interconnect (e.g., via a network card in promiscuous mode or via a mirrored switch port) in front of the reference server will not work—such tapping is susceptible to dropped packets in heavy traffic situations, which would violate this fundamental Tee assumption.

Synchronizing state on the SUT: Before RPC requests can be productively sent to the SUT, its state must be initialized such that its responses could be expected to match the reference server’s. For example, a file read’s responses won’t match unless the file’s contents are the same on both servers. Synchronizing the SUT’s state involves querying the reference server and updating the SUT accordingly.

For servers with large amounts of state, synchronizing can take a long time. Since only synchronized objects can be compared, few comparisons can be done soon after a SUT is inserted. Requests for objects that have yet to be synchronized produce no useful comparison

data. To combat this, the Tee could simply deny client requests until synchronization is complete. Then, when all objects have been synchronized, the Tee could relay and duplicate client requests knowing that they will all be for synchronized state. However, because we hope for the Tee to scale to terabyte- and petabyte-scale storage systems, complete state synchronization can take so long that denying client access would create significant downtime. To maintain acceptable availability, if a Tee is to be used for in situ testing, requests must be handled during initial synchronization even if they fail to yield meaningful comparison results.

Duplicating requests for the SUT: For RPC requests that can be serviced by the SUT (because the relevant state has been synchronized), the Tee needs to duplicate them, send them, and process the responses. This is often not as simple as just sending the same RPC request packets to the SUT, because IDs for the same object on the two servers may differ. For example, our NFS Tee must deal with the fact that the two file handles (reference server’s and SUT’s) corresponding to a particular file will differ; they are assigned independently by each server. During synchronization, any such ID mappings must be recorded for use during request duplication.

Comparing responses from the two servers: Comparing the responses from the reference server and SUT involves more than simple bitwise comparison. Each field of a response falls into one of three categories: bitwise-comparable, non-comparable, or loosely-comparable.

Bitwise-comparable fields should be identical for any correct server implementation. Most bitwise-comparable fields consist of data provided directly by clients, such as file contents returned by a file read.

Most non-comparable fields are either server-chosen values (e.g., cookies) or server-specific information (e.g., free space remaining). Differences in these fields do not indicate a problem, unless detailed knowledge of the internal meanings and states suggest that they do. For example, the disk space utilized by a file could be compared if both server’s are known to use a common internal block size and approach to space allocation.

Fields are loosely-comparable if comparing them requires more analysis than bitwise comparison—the reference and SUT values must be compared in the context of the field’s semantic meaning. For example, timestamps can be compared (loosely) by allowing differences small enough that they could be explained by clock skew, communication delay variation, and processing time variation.

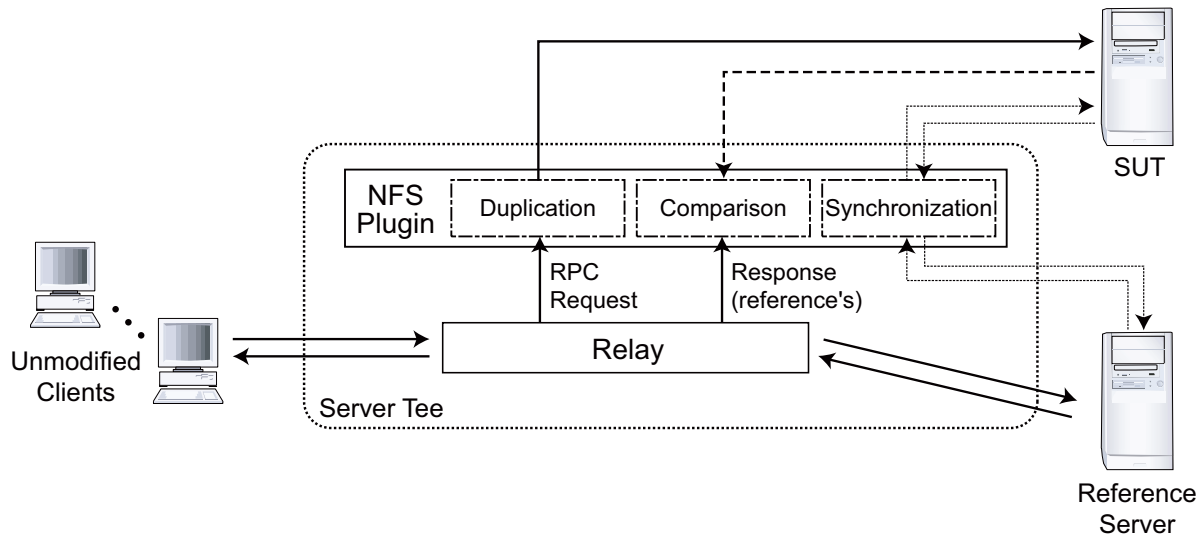


Figure 2: **Software architecture of an NFS Tee.** To minimize potential impact on clients, we separate the relaying functionality from the other three primary Tee functions (which contain the vast majority of the code). One or more NFS plug-ins can be dynamically initiated to compare a SUT to the reference server with which clients are interacting.

4 A NFSv3 Tee

This section describes the design and implementation of an NFSv3 Tee. It describes how components performing the four primary Tee tasks are organized and explains the architecture in terms of our design goals. It details nuanced aspects of state synchronization and response comparison, including some performance enhancements.

4.1 Goals and architecture

Our NFSv3 Tee’s architecture is driven by five design goals. First, we want to be able to use the Tee in live environments, which makes the reliability of the relay task crucial. Second, we want to be able to dynamically add a SUT and initiate comparison-based verification in a live environment.⁴ Third, we want the Tee to operate using reasonable amounts of machine resources, which pushes us to minimize runtime state and perform complex comparisons off-line in a post-processor. Fourth, we are more concerned with achieving a functioning, robust Tee than with performance, which guides us to have the Tee run as application-level software, acting as an active intermediary. Fifth, we want the comparison module to be flexible so that a user can customize of the rules to increase efficiency in the face of server idiosyncrasies that are understood.

Figure 2 illustrates the software architecture of our NFSv3 Tee, which includes modules for the four primary tasks. The four modules are partitioned into two

processes. One process relays communication between clients and the reference server. The other process (a “plug-in”) performs the three tasks that involve interaction with the SUT. The relay process exports RPC requests and responses to the plug-in process via a queue stored in shared memory. This two-process organization was driven by the first two design goals: (1) running the relay as a separate process isolates it from faults in the plug-in components, which make up the vast majority of the Tee code; (2) plug-ins can be started and stopped without stopping client interactions with the reference server.

When a plug-in is started, it attaches to the shared memory and begins its three modules. The synchronization module begins reading files and directories from the reference server and writing them to the SUT. As it does so, it stores reference server-to-SUT file handle mappings.

The duplication module examines each RPC request exported by the relay and determines whether the relevant SUT objects are synchronized. If so, an appropriate request for the SUT is constructed. For most requests, this simply involves mapping the file handles. The SUT’s response is passed to the comparison module, which compares it against the reference server’s response.

Full comparison consists of two steps: a configurable on-line step and an off-line step. For each mismatch found in the on-line step, the request and both responses are logged for off-line analysis. The on-line comparison rules are specified in a configuration file that describes how each response field should be compared. Off-line post-processing prunes the log of non-matching

⁴On a SUT running developmental software, developers may wish to make code changes, recompile, and restart the server repeatedly.

responses that do not represent true discrepancies (e.g., directory entries returned in different orders), and then assists the user with visualizing the “problem” RPCs. Off-line post-processing is useful for reducing on-line overheads as well as allowing the user to refine comparison rules without losing data from the real environment (since the log is a filtered trace).

4.2 State synchronization

The synchronization module updates the SUT to enable useful comparisons. Doing so requires making the SUT’s internal state match the reference server’s to the point that the two servers’ responses to a given RPC could be expected to match. Fortunately, NFSv3 RPCs generally manipulate only one or two file objects (regular files, directories, or links), so some useful comparisons can be made long before the entire file system is copied to the reference server.

Synchronizing an object requires establishing a point within the stream of requests where comparison could begin. Then, as long as RPCs affecting that object are handled in the same order by both servers, it will remain synchronized. The lifetime of an object can be viewed as a sequence of states, each representing the object as it exists between two modifications. Synchronizing an object, then, amounts to replicating one such state from the reference server to the SUT.

Performing synchronization offline (i.e., when the reference server is not being used by any clients) would be straightforward. But, one of our goals is the ability to insert a SUT into a live environment at runtime. This requires dealing with object changes that are concurrent with the synchronization process. The desire not to disrupt client activity precludes blocking requests to an object that is being synchronized. The simplest solution would be to restart synchronization of an object if a modification RPC is sent to the reference server before it completes. But, this could lead to unacceptably slow and inefficient synchronization of large, frequently-modified objects. Instead, our synchronization mechanism tracks changes to objects that are being synchronized. RPCs are sent to the reference server as usual, but are also saved in a *changeset* for later replay against the SUT.

Figure 3 illustrates synchronization in the presence of write concurrency. The state S1 is first copied from the reference server to the SUT. While this copy is taking place, a write (Wr1) arrives and is sent to the reference server. Wr1 is not duplicated to the SUT until the copy of S1 completes. Instead, it is recorded at the Tee. When the copy of S1 completes, a new write, Wr1’, is constructed based on Wr1 and sent to the SUT. Since no further concurrent changes need to be replayed, the object is marked

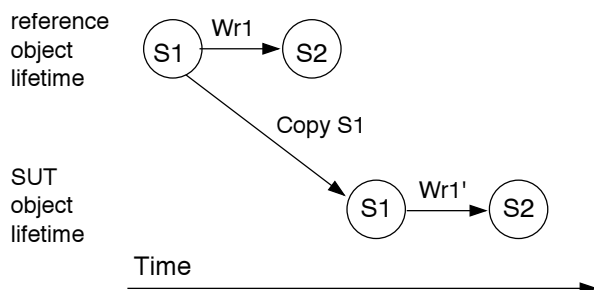


Figure 3: **Synchronization with a concurrent write.** The top series of states depicts a part of the lifetime of an object on the reference server. The bottom series of states depicts the corresponding object on the SUT. Horizontal arrows are requests executed on a server (reference or SUT), and diagonal arrows are full object copies. Synchronization begins with copying state S1 onto the SUT. During the copy of S1, write Wr1 changes the object on the reference server. At the completion of the copy of S1, the objects are again out of synchronization. Wr1’ is the write constructed from the buffered version of Wr1 and replayed on the SUT.

synchronized and all subsequent requests referencing it are eligible for duplication and comparison.

Even after initial synchronization, concurrent and overlapping updates (e.g., Wr1 and Wr2 in Figure 4) can cause a file object to become unsynchronized. Two requests are deemed overlapping if they both affect the same state. Two requests are deemed concurrent if the second one arrives at the relay before the first one’s response. This definition of concurrency accounts for both network reordering and server reordering. Since the Tee has no reliable way to determine the order in which concurrent requests are executed on the reference server, any state affected by both Wr1 and Wr2 is indeterminate. Resynchronizing the object requires re-copying the affected state from the reference server to the SUT. Since overlapping concurrency is rare, our Tee simply marks the object unsynchronized and repeats the process entirely.

The remainder of this section provides details regarding synchronization of files and directories, and describes some synchronization ordering enhancements that allow comparisons to start more quickly.

Regular file synchronization: A regular file’s state is its data and its attributes. Synchronizing a regular file takes place in three steps. First, a small unit of data and the file’s attributes are read from the reference server and written to the SUT. If a client RPC affects the object during this initial step, the step is repeated. This establishes a point in time for beginning the changeset. Second, the remaining data is copied. Third, any changeset entries are replayed.

A file’s changeset is a list of attribute changes and written-to extents. A bounded amount of the written data

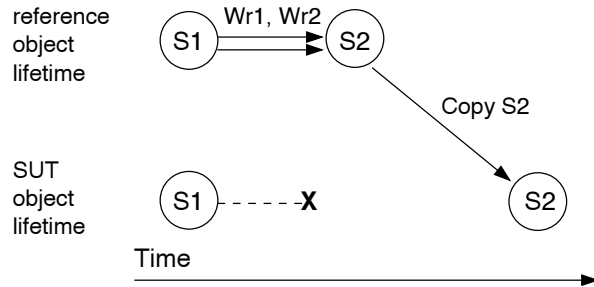


Figure 4: **Re-synchronizing after write concurrency.** The example begins with a synchronized object, which has state S1 on both servers. When concurrent writes are observed (Wr1 and Wr2 in this example), the Tee has no way of knowing their execution order at the reference server. As a consequence, it cannot know the resulting reference server state. So, it must mark the object as unsynchronized and repeat synchronization.

is cached. If more data was written, it must be read from the reference server to replay changes. As the changeset is updated, by RPCs to reference server, overlapping extents are coalesced to reduce the work of replaying them; so, for example, two writes to the same block will result in a single write to the SUT during the third step of file synchronization.

Directory synchronization: A directory's state is its attributes and the name and type of each of its children.⁵ This definition of state allows a directory to be synchronized regardless of whether its children are synchronized. This simplifies the tracking of a directory's synchronization status and allows the comparison of responses to directory-related requests well before the children are synchronized.

Synchronizing a directory is done by creating missing directory entries and removing extraneous ones. Hard links are created as necessary (i.e., when previously discovered file handles are found). As each unsynchronized child is encountered, it is enqueued for synchronization. When updates occur during synchronization, a directory's changeset will include new attribute values and two lists: entries to be created and entries to be removed. Each list entry stores the name, file handle, and type for a particular directory entry.

Synchronization ordering: By default, the synchronization process begins with the root directory. Each unknown entry of a directory is added to the list of files to be synchronized. In this way, the synchronization process works its way through the entire reference file system.

One design goal is to begin making comparisons as

⁵File type is not normally considered to be part of a directory's contents. We make this departure to facilitate the synchronization process. During comparison, file type is a property of the file, not of the parent directory.

quickly as possible. To accomplish this, our Tee synchronizes the most popular objects first. The Tee maintains a weighted moving average of access frequency for each object it knows about, identifying accesses by inspecting the responses to lookup and create operations. These quantities are used to prioritize the synchronization list. Because an object cannot be created until its parent directory exists on the SUT, access frequency updates are propagated from an object back to the file system root.

4.3 Comparison

The comparison module compares responses to RPC requests on synchronized objects. The overall comparison functionality proceeds in two phases: on-line and post-processed. The on-line comparisons are performed at runtime, by the Tee's comparison module, and any non-matching responses (both responses in their entirety) are logged together with the associated RPC request. The logged information allows post-processing to eliminate false non-matches (usually with more detailed examination) and to help the user to explore valid non-matches in detail.

Most bitwise-comparable fields are compared on-line. Such fields include file data, file names, soft link contents, access control fields (e.g., modes and owner IDs), and object types. Loosely-comparable fields include time values and directory contents. The former are compared on-line, while the latter (in our implementation) are compared on-line and then post-processed.

Directory contents require special treatment, when comparison fails, because of the looseness of the NFS protocol. Servers are not required to return entries in any particular order, and they are not required to return any particular number of entries in a single response to a REaddir or REaddirplus RPC request. Thus, entries may be differently-ordered and differently-spread across multiple responses. In fact, only when the Tee observes complete listings from both servers can some non-matches be definitively declared. Rather than deal with all of the resulting corner cases on-line, we log the observed information and leave it for the post-processor. The post-processor can link multiple RPC requests iterating through the same directory by the observed file handles and cookie values. It filters log entries that cannot be definitively compared and that do not represent mismatches once reordering and differing response boundaries are accounted for.

4.4 Implementation

We implemented our Tee in C++ on Linux. We used the State Threads user-level thread library. The relay runs

as a single process that communicates with clients and the reference server via UDP and with any plug-ins via a UNIX domain socket over which shared memory addresses are passed.

Our Tee is an active intermediary. To access a file system exported by the reference server, a client sends its requests to the Tee. The Tee multiplexes all client requests into one stream of requests, with itself as the client so that it receives all responses directly. Since the Tee becomes the source of all RPC requests seen by the reference server, it is necessary for the relay to map client-assigned RPC transaction IDs (XIDs) onto a separate XID space. This makes each XID seen by the reference server unique, even if different clients send requests with the same XID, and it allows the Tee to determine which client should receive which reply. This XID mapping is the only way in which the relay modifies the RPC requests.

The NFS plug-in contains the bulk of our Tee's functionality and is divided into four modules: synchronization, duplication, comparison, and the dispatcher. The first three modules each comprise a group of worker threads and a queue of lightweight request objects. The dispatcher (not pictured in Figure 2) is a single thread that interfaces with the relay, receiving shared memory buffers.

For each file system object, the plug-in maintains some state in a hash table keyed on the object's reference server file handle. Each entry includes the object's file handle on each server, its synchronization status, pointers to outstanding requests that reference it, and miscellaneous book-keeping information. Keeping track of each object consumes 236 bytes. Each outstanding request is stored in a hash table keyed on the request's reference server XID. Each entry requires 124 bytes to hold the request, both responses, their arrival times, and various miscellaneous fields. The memory consumption is untuned and could be reduced.

Each RPC received by the relay is stored directly into a shared memory buffer from the RPC header onward. The dispatcher is passed the addresses of these buffers in the order that the RPCs were received by the relay. It updates internal state (e.g., for synchronization ordering), then decides whether or not the request will yield a comparable response. If so, the request is passed to the duplication module, which constructs a new RPC based on the original by replacing file handles with their SUT equivalents. It then sends the request to the SUT.

Once responses have been received from both the reference server and the SUT, they are passed to the comparison module. If the comparison module finds any discrepancies, it logs the RPC and responses and optionally

alerts the user. For performance and space reasons, the Tee discards information related to matching responses, though this can be disabled if full tracing is desired.

5 Evaluation

This section evaluates the Tee along three dimensions. First, it validates the Tee's usefulness with several case studies. Second, it measures the performance impact of using the Tee. Third, it demonstrates the value of the synchronization ordering optimizations.

5.1 Systems used

All experiments are run with the Tee on an Intel P4 2.4GHz machine with 512MB of RAM running Linux 2.6.5. The client is either a machine identical to the Tee or a dual P3 Xeon 600MHz with 512MB of RAM running FreeBSD 4.7. The servers include Linux and FreeBSD machines with the same specifications as the clients, an Intel P4 2.2GHz with 512MB of RAM running Linux 2.4.18, and a Network Appliance FAS900 series filer. For the performance and convergence benchmarks, the client and server machines are all identical to the Tee mentioned above and are connected via a Gigabit Ethernet switch.

5.2 Case studies

An interesting use of the Tee is to compare popular deployed NFS server implementations. To do so, we ran a simple test program on a FreeBSD client to compare the responses of the different server configurations. The short test consists of directory, file, link, and symbolic link creation and deletion as well as reads and writes of data and attributes. No other filesystem objects were involved except the root directory in which the operations were done. Commands were issued at 2 second intervals.

Comparing Linux to FreeBSD: We exercised a setup with a FreeBSD SUT and a Linux reference server to see how they differ. After post-processing `REaddir` and `REaddirplus` entries, and grouping like discrepancies, we are left with the nineteen unique discrepancies summarized in Table 1. In addition to those nineteen, we observed many discrepancies caused by the Linux NFS server's use of some undefined bits in the `MODE` field (i.e., the field with the access control bits for owner, group, and world) of every file object's attributes. The Linux server encodes the object's type (e.g., directory, symlink, or regular file) in these bits, which causes the `MODE` field to not match FreeBSD's values in every response. To eliminate this recurring discrepancy, we modified the comparison rules to replace bitwise-comparison

Field	Count	Reason
EOF flag	1	FreeBSD server failed to return EOF at the end of a read reply
Attributes follow flag	10	Linux sometimes chooses not to return pre-op or post-op attributes
Time	6	Parent directory pre-op ctime and mtime are set to the current time on FreeBSD
Time	2	FreeBSD does not update a symbolic link's atime on READLINK

Table 1: Discrepancies when comparing Linux and FreeBSD servers. The fields that differ are shown along with the number of distinct RPCs for which they occur and the reason for the discrepancy.

of the entire MODE field with a loose-compare function that examines only the specification-defined bits.

Perhaps the most interesting discrepancy is the EOF flag, which is the flag that signifies that a read operation has reached the end of the file. Our Tee tells us that when a FreeBSD client is reading data from a FreeBSD server, the server returns FALSE at the end of the file while the Linux server correctly returns TRUE. The same discrepancy is observed, of course, when the FreeBSD and Linux servers switch roles as reference server and SUT. The FreeBSD client does not malfunction, which means that the FreeBSD client is not using the EOF value that the server returns. Interestingly, when running the same experiment with a Linux client, the discrepancy is not seen because the Linux client uses different request sequences. If a developer were trying to implement a FreeBSD NFS server clone, the NFS Tee would be an useful tool in identifying and properly mimicking this quirk.

The “attributes follow” flag, which indicates whether or not the attribute structure in the given response contains data,⁶ also produced discrepancies. These discrepancies mostly come from pre-operation directory attributes in which Linux, unlike FreeBSD, chooses not to return any data. Of course, the presence of these attributes represents additional discrepancies between the two servers’ responses, but the root cause is the same decision about whether to include the optional information.

The last set of interesting discrepancies comes from timestamps. First, we observe that FreeBSD returns incorrect pre-operation directory modification times (**mtime** and **ctime**) for the parent directory for RPCs that create a file, a hard link, or a symbolic link. Rather than the proper values being returned, FreeBSD returns the current time. Second, FreeBSD and Linux use different policies for updating the last access timestamp (**atime**). Linux updates the **atime** on the symlink file when the symlink is followed, whereas FreeBSD only updates the **atime** when the symlink file is accessed directly (e.g., by writing it’s value). This difference ex-

hibits discrepancies in RPCs that read the symlink’s attributes.

We also ran the test with the servers swapped (FreeBSD as reference and Linux as SUT). Since the client interacts with the reference server’s implementation, we were interested to see if the FreeBSD client’s interaction with a FreeBSD NFS server would produce different results when compared to the Linux server, perhaps due to optimizations between the like client and server. But, the same set of discrepancies were found.

Comparing Linux 2.6 to Linux 2.4: Comparing Linux 2.4 to Linux 2.6 resulted in very few discrepancies. The Tee shows that the 2.6 Kernel returns file metadata timestamps with nanosecond resolution as a result of its updated VFS layer, while the 2.4 kernel always returns timestamps with full second resolution. The only other difference we found was that the parent directory’s pre-operation attributes for SETATTR are not returned in the 2.4 kernel but are in the 2.6 kernel.

Comparing Network Appliance FAS900 to Linux and FreeBSD: Comparing the Network Appliance FAS900 to the Linux and FreeBSD servers yields a few interesting differences. The primary observation we are able to make is that the FAS900 replies are more similar to FreeBSD’s than Linux’s. The FAS900 handles its file MODE bits like FreeBSD without Linux’s extra file type bits. The FAS900, like the FreeBSD server, also returns all of the pre-operation directory attributes that Linux does not. It is also interesting to observe that the FAS900 clearly handles directories differently from both Linux and FreeBSD. The cookie that the Linux or FreeBSD server returns in response to a READDIR or READDIRPLUS call is a byte offset into the directory file whereas the Network Appliance filer simply returns an entry number in the directory.

Aside: It is interesting to note that, as an unintended consequence of our initial relay implementation, we discovered an implementation difference between the FAS900 and the Linux or FreeBSD servers. The relay modifies the NFS call’s XIDs so that if two clients happen to use the same XID, they don’t get mixed up when the Tee relays them both. The relay is using a sequence of values

⁶Many NFSv3 RPCs allow the affected object’s attributes to be included in the response, at the server’s discretion, for the client’s convenience.

for XIDs that is identical each time the relay is run. We found that, after restarting the Tee, requests would often get lost on the FAS900 but not on the Linux or FreeBSD servers. It turns out that the FAS900 caches XIDs for much longer than the other servers, resulting in dropped RPCs (as seeming duplicates) when the XID numbering starts over too soon.

Debugging the Ursa Major NFS server: Although the NFS Tee is new, we have started to use it for debugging an NFS server being developed in our group. This server is being built as a front-end to Ursa Major, a storage system that will be deployed at Carnegie Mellon as part of the Self-* Storage project [4]. Using Linux as a reference, we have found some non-problematic discrepancies (e.g., different choices made about which optional values to return) and one significant bug. The bug occurred in responses to the READ command, which never set the EOF flag even when the last byte of the file was returned. For the Linux clients used in testing, this is not a problem. For others, however, it is. Using the Tee exposed and isolated this latent problem, allowing it to be fixed proactively.

5.3 Performance impact of prototype

We use PostMark to measure the impact the Tee would have on a client in a live environment. We compare two setups: one with the client talking directly to a Linux server and one with the client talking to a Tee that uses the same Linux server as the reference. We expect a significant increase in latency for each RPC, but less significant impact on throughput.

PostMark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [6]. It creates a large number of small randomly-sized files (between 512 B and 9.77 KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append.

The experiments were done with a single client and up to sixteen concurrent clients. Except for the case of a single client, two instances of PostMark were run on each physical client machine. Each instance of PostMark ran with 10,000 transactions on 500 files and the biases for transaction types were equal. Except for the increase in the number of transactions, these are default PostMark values.

Figure 5 shows that using the Tee reduces client throughput when compared to a direct NFS mount. The reduction is caused mainly by increased latency due to the added network hop and overheads introduced by the fact

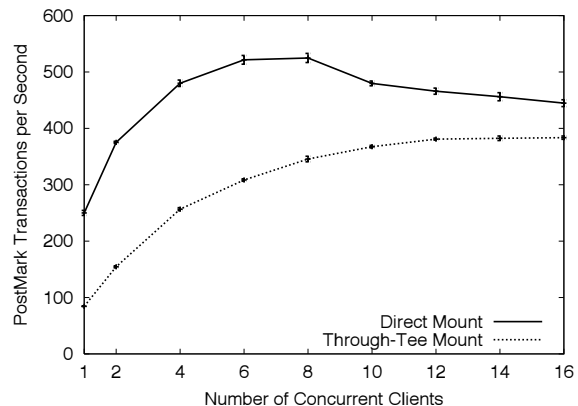


Figure 5: Performance with and without the Tee. The performance penalty caused by the Tee decreases as concurrency increases, because higher latency is the primary cost of inserting a Tee between client and reference server. Concurrency allows request propagation and processing to be overlapped, which continues to benefit the Through-Tee case after the Direct case saturates. The graph shows average and standard deviation of PostMark throughput, as a function of the number of concurrent instances.

that the Tee is a user-level process.

The single-threaded nature of PostMark allows us to evaluate both the latency and the throughput costs of our Tee. With one client, PostMark induces one RPC request at a time, and the Tee decreases throughput by 61%. As multiple concurrent PostMark clients are added, the percentage difference between direct NFS and through-Tee NFS performance shrinks. This indicates that the latency increase is a more significant factor than the throughput limitation—with high concurrency and before the server is saturated, the decrease in throughput drops to 41%. When the server is heavily loaded in the case of a direct NFS mount, the Tee continues to scale and with 16 clients the reduction in throughput is only 12%.

Although client performance is reduced through the use of the Tee, the reduction does not prevent us from using it to test synchronization convergence rates, do offline case studies, or test in live environments where lower performance is acceptable.

5.4 Speed of synchronization convergence

One of our Tee design goals was to support dynamic addition of a SUT in a live environment. To make such addition most effective, the Tee should start performing comparisons as quickly as possible. Recall that operations on a file object may be compared only if the object is synchronized. This section evaluates the effectiveness of the synchronization ordering enhancements described in Section 4.2. We expect them to significantly increase the speed with which useful comparisons can begin.

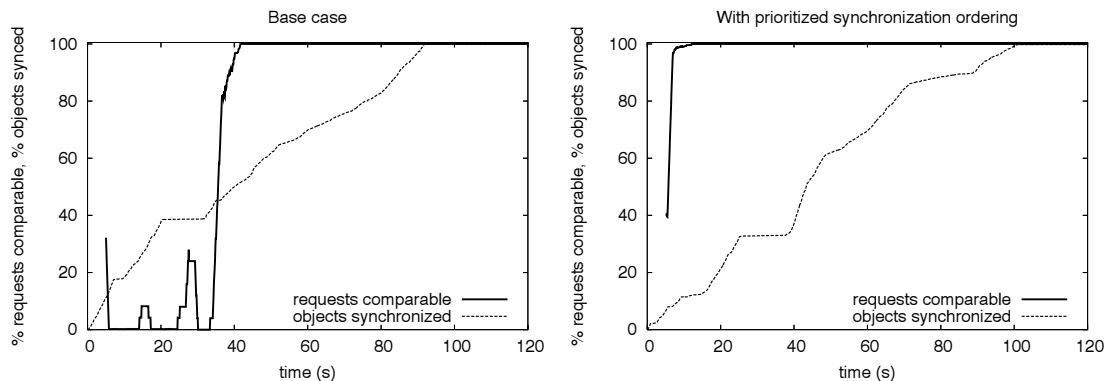


Figure 6: Effect of prioritized synchronization ordering on speed of convergence. The graph on the left illustrates the base case, with no synchronization ordering enhancements. The graph on the right illustrates the benefit of prioritized synchronization ordering. Although the overall speed with which the entire file system is synchronized does not increase (in fact, it goes down a bit due to contention on the SUT), the percentage of comparable responses quickly grows to a large value.

To evaluate synchronization, we ran an OpenSSH compile (the compile phase of the *ssh-build* benchmark used by Seltzer, et al. [12]) on a client that had mounted the reference server through the Tee. The compilation process was started immediately after starting the plugin. Both reference server and SUT had the same hardware configuration and ran the same version of Linux. No other workloads were active during the experiment. The OpenSSH source code shared a mount point with approximately 25,000 other files spread across many directories. The sum of the file sizes was 568MB.

To facilitate our synchronization evaluation, we instrumented the Tee to periodically write internal counters to a file. This mechanism provides us with two point-in-time values: the number of objects that are in a synchronized state and the total number of objects we have discovered thus far. It also provides us with two periodic values (counts within a particular interval): the number of requests enqueued for duplication to the SUT and the number of requests received by the plugin from the relay. These values allow us to compute two useful quantities. The first is the ratio of requests enqueued for duplication to requests received, expressed as a moving average; this ratio serves as a measure of the proportion of operations that were comparable in each time period. The second is the ratio of synchronized objects to the total number of objects in the file system; this value measures how far the synchronization process has progressed through the file system as a whole.

Figure 6 shows how both ratios grow over time for two Tee instances: one (on the left) without the synchronization ordering enhancements and one with them. Although synchronization of the entire file system requires over 90 seconds, prioritized synchronization ordering quickly enables a high rate of comparable responses.

Ten seconds into the experiment, almost all requests produced comparable responses with the enhancements. Without the enhancements, we observe that a high rate of comparable responses is reached at about 40 seconds after the plugin was started. The rapid increase observed in the unoptimized case at that time can be attributed to the synchronization module reaching the OpenSSH source code directory during its traversal of the directory tree.

The other noteworthy difference between the unordered case and the ordered case is the time required to synchronize the entire file system. Without prioritized synchronization ordering, it took approximately 90 seconds. With it, this figure was more than 100 seconds. This difference occurs because the prioritized ordering allows more requests to be compared sooner (and thus duplicated to the SUT), creating contention for SUT resources between synchronization-related requests and client requests. The variation in the rate with which objects are synchronized is caused by a combination of variation in object size and variation in client workload (which contends with synchronization for the reference server).

6 Discussion

This section discusses several additional topics related to when comparison-based server verification is useful.

Debugging FS client code: Although its primary *raison d'être* is file server testing, comparison-based FS verification can also be used for diagnosing problems with client implementations. Based on prior experiences, we believe the best example of this is when a client is observed to work with some server implementations and not others (e.g., a new version of a file server). Detailed insight can be obtained by comparing server responses to

request sequences with which there is trouble, allowing one to zero in on what unexpected server behavior the client needs to cope with.

Holes created by non-comparable responses: Comparison-based testing is not enough. Although it exposes and clarifies some differences, it is not able to effectively compare responses in certain situations, as described in Section 4. Most notably, concurrent writes to the same data block are one such situation—the Tee cannot be sure which write was last and, therefore, cannot easily compare responses to subsequent reads of that block. Note, however, that most concurrency situations can be tested.

More stateful protocols: Our file server Tee works for NFS version 3, which is a stateless protocol. The fact that no server state about clients is involved simplifies Tee construction and allows quick ramp up of the percentage of comparable operations. Although we have not built one, we believe that few aspects would change significantly in a file server Tee for more stateful protocols, such as CIFS, NFS version 4, and AFS [5]. The most notable change will be that the Tee must create duplicate state on the SUT and include callbacks in the set of “responses” compared—callbacks are, after all, external actions taken by servers usually in response to client requests. A consequence of the need to track and duplicate state is that comparisons cannot begin until both synchronization completes **and** the plug-in portion of the Tee observes the beginnings of client sessions with the server. This will reduce the speed at which the percentage of comparable operations grows.

7 Related work

On-line comparison has a long history in computer fault-tolerance [14]. Usually, it is used as a voting mechanism for determining the right result in the face of problems with a subset of instances. For example, the triple modular redundancy concept consists of running multiple instances of a component in parallel and comparing their results; this approach has been used, mainly, in very critical domains where the dominant fault type is hardware problems. Fault-tolerant consistency protocols (e.g., Paxos [11]) for distributed systems use similar voting approaches.

With software, deterministic programs will produce the same answers given the same inputs, so one accrues little benefit from voting among multiple instances of the same implementation. With multiple implementations of the same service, on the other hand, benefits can accrue. This is generally referred to as N-version programming [2]. Although some argue that N-version program-

ming does not assist fault-tolerance much [8, 9], we view comparison-based verification as a useful application of the basic concept of comparing one implementation’s results to those produced by an independent implementation.

One similar use of inter-implementation comparison is found in the Ballista-based study of POSIX OS robustness [10]. Ballista [3] is a tool that exercises POSIX interfaces with various erroneous arguments and evaluates how an OS implementation copes. In many cases, DeVale, et al. found that inconsistent return codes were used by different implementations, which clearly creates portability challenges for robustness-sensitive applications.

Use of a server Tee applies the proxy concept [13] to allow transparent comparison of a developmental server to a reference server. Many others have applied the proxy concept for other means. In the file system domain, specifically, some examples include Slice [1], Zforce [17], Cuckoo [7], and Anypoint [16]. These all interpose on client-server NFS activity to provide clustering benefits to unmodified clients, such as replication and load balancing. Most of them demonstrate that such interposing can be done with minimal performance impact, supporting our belief that the slowdown of our Tee’s relaying could be eliminated with engineering effort.

8 Summary

Comparison-based server verification can be a useful addition to the server testing toolbox. By comparing a SUT to a reference server, one can isolate RPC interactions that the SUT services differently. If the reference server is considered correct, these discrepancies are potential bugs needing exploration. Our prototype NFSv3 Tee demonstrates the feasibility of comparison-based server verification, and our use of it to debug a prototype server and to discover interesting discrepancies among production NFS servers illustrates its usefulness.

Acknowledgements

We thank Raja Sambasivan and Mike Abd-El-Malek for help with experiments. We thank the reviewers, including Vivek Pai (our shepherd), for constructive feedback that improved the presentation. We thank the members and companies of the PDL Consortium (including EMC, Engenio, Hewlett-Packard, HGST, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the National Science Foun-

dation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620-01-1-0433, and by the Army Research Office, under agreement number DAAD19-02-1-0389.

References

- [1] D. C. Anderson, J. S. Chase, and A. M. Vahdat. Interposed request routing for scalable network storage. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 22–25 October 2000), 2000.
- [2] L. Chen and A. Avizienis. N-version programming: a fault tolerance approach to reliability of software operation. *International Symposium on Fault-Tolerant Computer Systems*, pages 3–9, 1978.
- [3] J. P. DeVale, P. J. Koopman, and D. J. Guttendorf. The Ballista software robustness testing service. *Testing Computer Software Conference* (Bethesda, MD, 14–18 June 1999). Unknown publisher, 1999.
- [4] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. *Self-* Storage: Brick-based storage with automated administration*. Technical Report CMU-CS-03-178. Carnegie Mellon University, August 2003.
- [5] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, **6**(1):51–81. ACM, February 1988.
- [6] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.
- [7] A. J. Klosterman and G. Ganger. *Cuckoo: layered clustering for NFS*. Technical Report CMU-CS-02-183. Carnegie Mellon University, October 2002.
- [8] J. C. Knight and N. G. Leveson. A reply to the criticisms of the Knight & Leveson experiment. *ACM SIGSOFT Software Engineering Notes*, **15**(1):24–35. ACM, January 1990.
- [9] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumptions of independence in multiversion programming. *Transactions on Software Engineering*, **12**(1):96–109, March 1986.
- [10] P. Koopman and J. DeVale. Comparing the robustness of POSIX operating systems. *International Symposium on Fault-Tolerant Computer Systems* (Madison, WI, 15–18 June 1999), 1999.
- [11] L. Lamport. Paxos made simple. *ACM SIGACT News*, **32**(4):18–25. ACM, December 2001.
- [12] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. *USENIX Annual Technical Conference* (San Diego, CA, 18–23 June 2000), pages 71–84, 2000.
- [13] M. Shapiro. Structure and encapsulation in distributed systems: the proxy principle. *International Conference on Distributed Computing Systems* (Cambridge, Mass), pages 198–204. IEEE Computer Society Press, Catalog number 86CH22293-9, May 1986.
- [14] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems: design and evaluation*. Digital Press, Second edition, 1992.
- [15] SPEC SFS97_R1 V3.0 benchmark, Standard Performance Evaluation Corporation, August, 2004. <http://www.specbench.org/sfs97r1/>.
- [16] K. G. Yocum, D. C. Anderson, J. S. Chase, and A. M. Vahdat. Anypoint: extensible transport switching on the edge. *USENIX Symposium on Internet Technologies and Systems* (Seattle, WA, 26–28 March 2003), 2003.
- [17] Z-force, Inc., 2004. www.zforce.com.

Active Internet Traffic Filtering: Real-Time Response to Denial-of-Service Attacks

Katerina Argyraki David R. Cheriton
Distributed Systems Group
Stanford University
{argyraki, cheriton}@dsg.stanford.edu

Abstract

This paper describes *Active Internet Traffic Filtering* (AITF), a mechanism for blocking highly distributed denial-of-service (DDoS) attacks. These attacks are an acute contemporary problem, with few practical solutions available today; we describe in this paper the reasons why no effective DDoS filtering mechanism has been deployed yet. We show that the current Internet's routers have sufficient filtering resources to thwart such attacks, with the condition that attack traffic be blocked close to its sources; AITF leverages this observation. Our results demonstrate that AITF can block a million-flow attack within seconds, while it requires only tens of thousands of wire-speed filters per participating router — an amount easily accommodated by today's routers. AITF can be deployed incrementally and yields benefits even to the very first adopters.

1 Introduction

We have recently witnessed a dramatic increase in the frequency and ferocity of distributed denial-of-service (DDoS) attacks. In December 2003, an attack kept SCO's web site practically unreachable for more than a day [6]; in June 2004, another attack flooded Akamai's name servers, disrupting access to its clients for 2 hours, including the Google and Yahoo search engines [7]; a month later, an attack flooded DoubleClick's name servers, disabling ad distribution to its 900 clients for 3 hours [8]. Considering that network downtime costs hundreds of thousands of dollars per hour [19], such incidents can translate into millions of dollars of lost revenue for the victim. Yet, the DDoS problem remains unsolved.

We recognize three (not all of them orthogonal) problems that render DDoS traffic hard to filter:

Source address spoofing: An attack source often uses multiple fake source IP addresses to send its traffic. As a result, the victim can neither identify the attack source nor specify a filtering rule (e.g., “block all traffic with

source IP address S ”) that selectively blocks its traffic.

Large number of attack sources: Each hardware router has only a limited number of filters that can block traffic without degrading the router's performance (i.e., filters operating at wire speed). The limitation comes from cost and space. Wire-speed filters are typically stored in expensive TCAM (Ternary Content Addressable Memory), which they share with the router's forwarding table. Some of the largest TCAM chips available today accommodate 256K entries [4]. A sophisticated router linecard fits at most 1 TCAM chip [1], i.e., tens of thousands of filters per network interface. On the other hand, a large-scale attack can involve millions of attack sources [22]. So, even if source address spoofing were completely eliminated, i.e., even if the victim could identify each attack source and specify a filtering rule for it, the victim's firewall would not have enough filters to accommodate all the rules.

The straightforward solution to such a resource problem is aggregation: Don't install a separate filter for each attack source; instead, identify the IP prefixes that cover most attack sources and block all traffic from these prefixes. Unfortunately, filter aggregation does not work in most DDoS scenarios: Attack sources are typically worm-infected populations, highly distributed across the Internet. As a result, blocking the prefixes that correspond to the attack sources results in blocking most Internet prefixes, thereby causing severe collateral damage.

Pushing filtering into the Internet core does not scale:

If the victim's firewall cannot block attack traffic by itself, the straightforward solution is to push filtering of attack traffic back into the Internet core: Identify the upstream peering networks that forward attack traffic and send them appropriate filtering requests. Unfortunately, this approach does not scale, because it introduces end-to-end filtering state into core routers. Consider a core router receiving filtering requests from 10 victims; each victim is under attack by a million sources. The core router can either block traffic from each attack source to each victim, or aggregate filtering rules and rate-limit

all traffic going to the victims. The former requires 10 million filters; the latter requires only 10 filters, but sacrifices most good traffic going to the victims.

Yet, there *are* enough filtering resources in the Internet to block such large-scale attacks. An attack coming from thousands of different networks involves thousands of routers; assuming each router contributes a few thousand filters, there are millions of filters available to block attack traffic. And the closer we get to the attack sources, the larger the amount of filtering resources available per attack source — it is the victim’s firewall and the Internet core that are the “filtering bottleneck”. Unfortunately, today, the victim has no access to these resources, since there is no way for a DDoS victim to identify routers located close to the attack sources and make them block attack traffic.

In this paper, we present a DDoS filtering mechanism that overcomes these problems. Our source address spoofing solution is a hardware-friendly variant of the IP route record (RR) technique [20]. Although different from traditional packet marking techniques [21, 14] (which do not provide an explicit recorded route in each packet), our RR approach is not a radical departure from them, either. Our main contribution is Active Internet Traffic Filtering (AITF), a protocol that leverages recorded route information to block attack traffic.

An AITF-enabled receiver uses the routes recorded on incoming packets to identify the last point of trust on each attack path and causes attack traffic to be blocked at that point, i.e., as close as possible to its sources. We provide a way to do this securely — AITF prevents abuse by malicious nodes seeking to disrupt other nodes’ communications. We show that our approach can selectively block a million attack sources, yet requires only tens of thousands of TCAM memory entries and a few megabytes of DRAM memory from each participating router; these numbers correspond to the specifications of real products [4, 1]. We also provide an incremental deployment scenario, in which even early adopters receive a concrete benefit; this benefit is compounded by further deployment.

The rest of the paper is organized as follows: Section 2 describes route record and how a receiver can use it to identify distinct traffic flows. Section 3 describes the AITF protocol in detail, naively assuming that no source address spoofing occurs. We remove this assumption in Section 4, where we describe how AITF deals with spoofing attacks. We estimate AITF performance in Section 5 and verify our estimates through simulation in Section 6. Section 7 analyzes deployment issues, Section 8 discusses additional attacks and potential defenses, and Section 9 presents related work. Section 10 concludes the paper.

2 Limiting spoofing

2.1 Route Record

A router that participates in a route record (RR) scheme writes its IP address on each packet it forwards. In our approach, only border routers participate in RR. As a result, each packet carries the identities of a sub-list of the border routers that forwarded it. For example, in Figure 1, border routers A_{gw} , X , Y , and V_{gw} are RR-enabled; each packet sent by host A to host V carries recorded route $\{A_{gw} X Y V_{gw}\}$ upon reaching its destination.

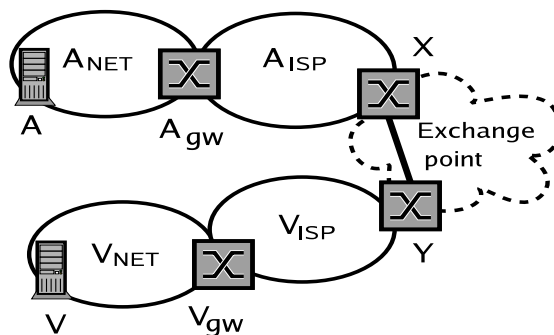


Figure 1: Packets sent by host A to host V carry recorded route $\{A_{gw} X Y V_{gw}\}$.

We implement RR functionality as a “shim” protocol between the IP and transport layers, i.e., the RR header is introduced as the beginning of the IP payload. We chose not to use the traditional IP RR option, because of its performance overhead — traditional IP RR packets are typically handled by routers off the fast path. The details of how each participating router adds its address to the RR header are described in the Appendix, Section A.1.

When a packet crosses an RR-enabled, “non-malicious” Internet area adjacent to its destination domain, its recorded route includes an authentic (non-spoofed) suffix. Specifically, the last n components of the recorded route are authentic, when the last n border routers crossed by the packet are RR-enabled, and there is no malicious node on the path that interconnects them. As we explain next, this enables a receiver to identify distinct incoming traffic flows in the face of source address spoofing.

2.2 Identifying Distinct Flows

We define the *recorded path* of a packet as the sequence of IP addresses that correspond to: the packet’s source, the list of border routers specified in the RR header, and the packet’s destination. We define a *flow* as the set of all packets that share a common recorded path suffix. For example, in Figure 1, all packets with path $\{A A_{gw} X Y V_{gw} V\}$ constitute a distinct flow; all packets with path $\{* A_{gw} X Y V_{gw} V\}$ also constitute a dis-

tinct (aggregate) flow. We use $F\{P\}$ to refer to flow F with recorded path P .

A DDoS victim feeds recorded paths into a local policy module, which classifies incoming traffic in distinct flows, decides which ones are undesired and forms filtering requests against them. The operation of the policy module depends on the specific service run by the victim and is outside the scope of this paper. We call it a “policy module”, because it determines a “defense policy”, i.e., which flows must be blocked. AITF (described in the next section) is the mechanism that enforces the chosen policy.

It is up to the policy module to classify incoming traffic in multiple “flow levels”, in order to identify undesired flows in the face of source address and path spoofing. For example, consider that in Figure 1 attack source A is sending high-rate traffic to victim V . If network A_{NET} prevents source address spoofing, V can easily identify $F_1\{A A_{gw} X Y V_{gw} V\}$ as a high-rate flow and, thus, undesired. If A is able to spoof multiple source IP addresses, V can only identify $F_2\{* A_{gw} X Y V_{gw} V\}$ as the undesired flow.

Once the policy module identifies an undesired flow, it sends a filtering request to the local AITF process. AITF does not assume that the recorded path of an undesired flow coincides with its real path. I.e., if the policy module identifies $F\{* A_{gw} X Y V_{gw} V\}$ as an undesired flow, this only means that the victim does not want to receive any more packets with this recorded path; it does not necessarily mean that these packets are indeed forwarded by A_{gw} , X , and Y .

3 Basic AITF Protocol

We start with an overview of the protocol (Section 3.1) and terminology (Section 3.2); we describe our algorithm incrementally, through Sections 3.3, 3.4, 3.5, and 3.6; we discuss appropriate values for its parameters in Section 3.7. To simplify description, we naively assume that no source address/path spoofing occurs — we remove this assumption in the next section.

3.1 Overview

Upon identifying an undesired flow, the victim sends a filtering request to its gateway (V_{gw} in Figure 1). The victim’s gateway temporarily blocks the undesired flow and identifies the border router located closest to the attack source(s) — call it the *attack gateway* (A_{gw} in Figure 1). Then, the victim’s gateway initiates a “counter-connection” setup with the attack gateway, i.e., an agreement not to transmit certain packets — the opposite of a TCP connection setup, which is an agreement to exchange packets. As soon as the counter-connection setup

is completed, the victim’s gateway can remove its temporary filter. If the attack gateway does not cooperate, the victim’s gateway can *escalate* the filtering request to the next border router closest to the attack gateway (X in Figure 1). Escalation can continue recursively until a router along the attack path responds and a counter-connection setup is completed. If no router responds, attack traffic is blocked locally by the victim’s gateway. However, as we will see, AITF both assists and motivates routers close to the attack source(s) to help block attack traffic.

3.2 Terminology

The recorded path P of an undesired flow has form $\{A A_{gw} \dots V_{gw} V\}$, where

- A is the “attack source”, i.e., the node thought to be generating the undesired traffic; if $A = *$, all traffic through A_{gw} is undesired.
- A_{gw} is the “attack gateway”, i.e., the border router thought to be closest to A .
- V_{gw} is the “victim’s gateway”, i.e., the border router closest to the victim.
- V is the victim.

We assume that the only node affected by the attack is V ; e.g., if this is a flooding attack, the only part of the network that is congested is the tail-circuit from V_{gw} to V . If V_{gw} were also affected, it itself would be the “victim”, and its closest upstream border router would be the “victim’s gateway”.

3.3 Blocking Close to the Attack Source

As shown in Figure 2, AITF involves 4 entities:

1. The victim V sends a filtering request to V_{gw} , specifying an undesired flow F .
2. The victim’s gateway V_{gw} :
 - (a) Installs a temporary filter to block F for T_{tmp} seconds.
 - (b) Initiates a 3-way *handshake* with A_{gw} .
 - (c) Removes its temporary filter, upon completion of the handshake.
3. The attack gateway A_{gw} :
 - (a) Responds to the 3-way handshake.
 - (b) Installs a temporary filter to block F for T_{tmp} seconds, upon completion of the handshake.
 - (c) Sends a filtering request to the attack source A , to stop F for $T_{long} \gg T_{tmp}$ minutes.

- (d) Removes its temporary filter, if A complies within T_{tmp} seconds; otherwise, it disconnects A .

4. The attack source A stops F for T_{long} minutes or risks disconnection.

All filtering requests are rate limited. I.e., the victim's gateway accepts a limited rate of requests from each alleged victim. Similarly, the attack gateway (i) accepts a limited rate of requests (handshake initializations) from each alleged victim gateway and (ii) sends a limited rate of requests to each alleged attack source.

The reason for the temporary filter on the victim's gateway is to immediately protect the victim until the attack gateway takes responsibility. The reason for directly contacting the attack gateway is to avoid creating a filtering bottleneck in the Internet core. Finally, the reason for the 3-way handshake is to enable the attack gateway to verify that the requester of the filter is indeed on the path to the alleged victim; the handshake is further explained next.

3.4 Securing Edge-to-edge Communication

The 3-way handshake is depicted in Figure 2: V_{gw} sends to A_{gw} a request to block F ; A_{gw} sends to V a message that includes F and a nonce; V_{gw} intercepts the message and sends it back to A_{gw} .

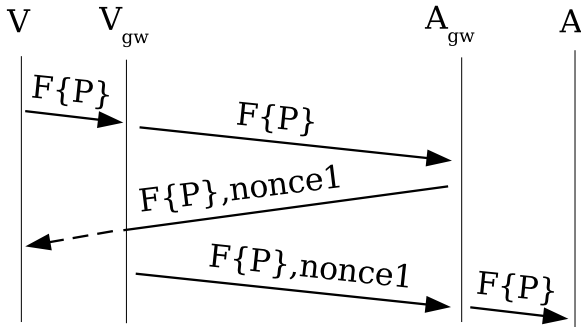


Figure 2: AITF entities and message exchange.

V_{gw} proves its location on the path to V by intercepting the nonce sent to V . This prevents malicious node M , located off the path from A_{gw} to V_{gw} , from causing a filter to be installed at A_{gw} and block traffic to V . By picking a sufficiently large and properly random value for the nonce, it can be made arbitrarily difficult for M to guess it (see Section 5.4).

To avoid buffering state on incomplete 3-way handshakes, A_{gw} computes the nonce as follows:

$$nonce1 = hash_{key}(F)$$

where key is a local key and $hash$ is a keyed hash function. To verify the authenticity of a completion message,

A_{gw} just hashes the flow included in the message and compares the result to the nonce included in the message (similar to the TCP SYN-cookie technique [13]).

3.5 Identifying Liars

With what we have described so far, there are two entities that can lie: (i) An attack source can pause an undesired flow (to avoid disconnection) and resume as soon as the attack gateway has removed its temporary filter. (ii) An attack gateway can pause an undesired flow and resume as soon as the victim's gateway has removed its temporary filter. To catch such liars, we introduce the *shadow filtering table*.

Every time a gateway removes a temporary filter from its TCAM, it creates a copy in DRAM that expires after T_{long} . This “shadow filter” helps check whether the corresponding undesired flow is released prematurely (before T_{long}) by its source. For example, suppose attack gateway A_{gw} has already told attack source A to block F ; now suppose A_{gw} receives a new filtering request against F and installs a new temporary filter; if the new filter catches F traffic, A_{gw} checks its shadow filtering table, finds out that a shadow filter for F already exists (i.e., A has already been told to stop once) and disconnects A .

The victim's gateway uses the same technique to check whether the attack gateway keeps the undesired flow blocked for T_{long} minutes. The only difference is that the attack gateway has to be caught violating the filtering agreement twice to be classified as “lying” — the first time could be due to a lying attack source, so the attack gateway is given the benefit of the doubt once.

3.6 Dealing with Non-Cooperative Gateways

An attack gateway can deal with a non-cooperative attack source by disconnecting it. This is possible because the attack gateway is the border router providing Internet connectivity to the attack source. The victim's gateway can obviously not deal with a non-cooperative attack gateway the same way, since they belong to separate (not even peering) administrative domains. To address this, we introduce *escalation*.

An attack gateway is classified as “non-cooperative”, if it does not respond to the handshake or responds, but is caught violating the filtering agreement twice. In that case, the victim's gateway can “escalate” the filtering request to the border router that follows the non-cooperative attack gateway on the flow's path. The new attack gateway is requested to block all traffic from the last non-cooperative attack gateway to the victim. For example, in Figure 1, V_{gw} first contacts A_{gw} asking it to block all traffic from A to V . If A_{gw} does not cooperate,

V_{gw} can contact X asking it to block all traffic from A_{gw} to V . Simply said, an attack gateway either cooperates and blocks traffic from its misbehaving client(s) to the victim, or risks losing its connectivity to the victim overall. This is a strong incentive to cooperate, especially when the victim is a popular public-access site like eBay or Amazon.

There are cases in which escalation is good, and cases in which it is bad. E.g., if eBay is losing most of its good traffic to a severe flooding attack, it makes sense to block access from non-cooperative attack gateways in order to preserve connectivity to the rest of the world. On the other hand, if eBay wants to get rid of a couple of annoying AOL clients, it does not make sense to block the entire AOL, even if its gateway does not cooperate. Deciding whether escalation is a good or a bad idea is the responsibility of the policy module. The policy module communicates its decision by declaring (or not) an undesired flow “escalatable”.

The victim’s gateway escalates a filtering request if and only if (i) the corresponding flow is escalatable and (ii) local filter utilization has exceeded a pre-configured threshold. Otherwise, the undesired flow is blocked locally for T_{long} minutes.

A smart attack source can introduce multiple fake components in the beginning of the RR header and make the victim’s gateway escalate multiple times (as many as the fake components), before it actually contacts an authentic border router. To avoid such abuse, a DDoS victim can simply ignore the first components of “suspiciously” long paths when classifying flows. For example, 95% of Internet domains are no more than 6 hops apart [17]. So, a DDoS victim can consider only the last 6 components of the RR header when classifying flows. This limits the number of unsuccessful escalation attempts to $6 - n$, where n is the number of RR/AITF-enabled, cooperative border routers on the attack path. However, this also sacrifices the traffic of good sources collocated with bad sources, in networks that are more than 6 domains away from the victim. The policy module must decide whether keeping this traffic is worth the unsuccessful escalation attempts.

3.7 Filter Timeout Values

The goal of a temporary filter on the victim’s gateway is to block an undesired flow until the corresponding handshake is complete. Considering that Internet round-trip times range from 50 to 200 msec, a “safe” value is $T_{tmp} = 1$ sec.

The choice of the long-term filter timeout T_{long} involves the following trade-off: An attack source A is typically an “innocent” end-host compromised through a worm. A large T_{long} of, say, 30 minutes guarantees that

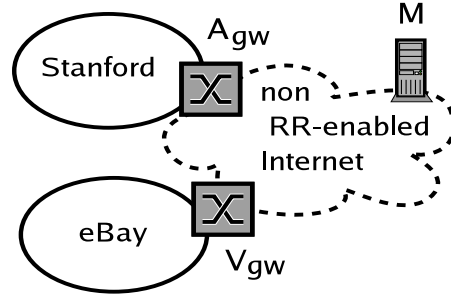


Figure 3: Malicious node M pretends to be at Stanford and sends undesired traffic to eBay; its packets carry (spoofed) recorded path $\{* A_{gw} V_{gw} eBay\}$.

the victim V will not receive any undesired traffic from the corresponding attack source A for at least 30 minutes; on the other hand, it also guarantees that V will not receive any traffic at all from A , even if A is appropriately patched before T_{long} expires. Section 5 explains how exactly the value for T_{long} affects AITF performance.

4 Adding Resistance to Spoofing

As presented thus far, AITF is effective only if deployed in a significant portion of the Internet, adjacent to the victim’s network. Otherwise, a malicious node can use path spoofing and abuse AITF to disrupt other nodes’ communications. We illustrate with an example.

Figure 3 illustrates an early deployment stage, where only eBay and Stanford have deployed RR and AITF. Traffic sent by Stanford hosts to eBay has path $\{* A_{gw} V_{gw} eBay\}$. Malicious node M spoofs this path and sends to eBay high-rate traffic that appears to be coming from Stanford. V_{gw} requests from A_{gw} to stop; although A_{gw} agrees, V_{gw} continues to see high-rate traffic from Stanford; it falsely concludes that A_{gw} is non-cooperative and blocks all traffic from Stanford to eBay.

To prevent this abuse, we augment the recorded path with randomized components. Now each RR-enabled border router that forwards a packet writes on the packet (i) its IP address and (ii) a random value that depends on the packet’s destination. By picking a sufficiently large and properly random value, it can be made arbitrarily hard for malicious off-the-path nodes to guess it (see Section 5.4). For example, in Figure 3, traffic sent by Stanford to eBay, has recorded path $\{* A_{gw}:R_1 V_{gw}:R_2 eBay\}$, where R_1 and R_2 are random values inserted by A_{gw} and V_{gw} respectively. To avoid keeping per destination state, each router computes the random value to insert in each packet as follows:

$$R = \text{hash}_{key}(D)$$

where key is a local key, $hash$ is a keyed hash function

and D is the packet's destination.

When A_{gw} receives a request to block undesired flow $F\{P\}$, it checks whether it indeed forwarded F , i.e., whether P includes the correct random value. If yes, A_{gw} commits to filter F by responding to the handshake as described in Section 3.4. Otherwise, A_{gw} responds with the “authentic” path P' , i.e., the path that includes the correct random value. So, if the victim's gateway sends a filtering request with a spoofed path, the handshake consists of just two messages, depicted in Figure 4.

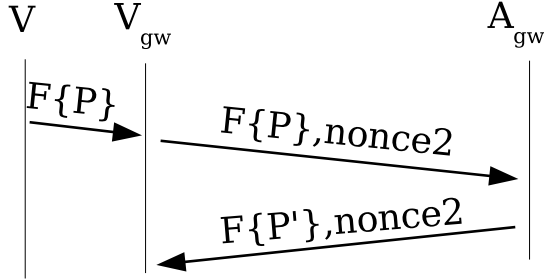


Figure 4: V_{gw} sends a filtering request against spoofed flow $F\{P\}$ that appears to be coming from A_{gw} ; A_{gw} responds with the authentic path P' , which includes the correct random value inserted by A_{gw} in all packets addressed to V . We added one more nonce, to enable V_{gw} to verify that the response with the authentic path is indeed coming from A_{gw} .

The victim's gateway uses the authentic path P' to recognize spoofed traffic that claims to be coming from A_{gw} and block it. For example, in Figure 3, V_{gw} blocks everything that appears to be forwarded by A_{gw} but does not include A_{gw} 's correct random value R_1 , i.e., V_{gw} blocks traffic with path $\{* A_{gw}!R_1 V_{gw} eBay\}$. Note that in the specific example, V_{gw} can only block this traffic locally — escalation is not an option, since no other domain has deployed AITF.

We defer answering the following questions to later sections: (i) Exactly how large must each random component be? (ii) A router computes its random values based on a local key; how often does this key expire? (iii) When the local key changes, all random values communicated to victim gateways as part of the handshake become invalid; does the router notify the corresponding victim gateways?

5 Evaluation

In this section, we use back-of-the-envelope calculations to show that (i) a victim can have an undesired flow blocked within milliseconds; (ii) the victim's gateway can block a certain number of undesired flows with two orders of magnitude fewer filters than flows and a reasonable amount of DRAM; and (iii) the probability of abusing AITF can be made arbitrarily low.

5.1 Filtering Response Time

We define *filtering response time* T_{fr} as the time that elapses from the moment the victim sends a filtering request against an undesired flow, until the victim stops receiving the flow. With AITF, this is equal to the one-way delay from the victim to the victim's gateway plus the negligible overhead of processing the filtering request and installing the corresponding temporary filter. I.e., filtering response time is a few milliseconds.

If there are compromised routers on the flow's path, they can agree to filter the flow and later break the agreement — recall that each attack gateway is given two chances to cooperate. This results in the victim receiving “spikes” of the undesired flow after T_{fr} . Spikes are spaced out by at least T_{tmp} seconds; the number of spikes is bounded from above by $2 \times n$, where n is the number of compromised routers on the flow's path. The effect of these spikes on the victim is insignificant, as we demonstrate with later simulation results.

5.2 Filtering Rate, Capacity and Gain

In this section, we examine three basic AITF properties: how much attack traffic it blocks, how fast it does so, and at what cost. To quantify these properties, we use three simple metrics: the *filtering rate* of a router is the number of flows that the router can block every second; the *filtering capacity*, is the number of flows that the router can keep blocked simultaneously; the *filtering gain* is the number of blocked flows divided by the required number of filters:

$$G = \frac{N_{flows}}{N_{filters}}$$

For example, $G = 1$ flow/filter means that the router uses 1 filter to keep 1 flow blocked.

Victim's gateway — best-case scenario: Assume all attack gateways cooperate and all attack sources comply with their gateways to avoid disconnection. In this case, every time V_{gw} satisfies 1 filtering request, it spends 1 filter for T_{tmp} seconds and causes 1 flow to be blocked for T_{long} minutes. Thus, if V_{gw} uses $N_{filters}$ filters, it achieves filtering rate $\frac{N_{filters}}{T_{tmp}}$ flows/sec, filtering capacity $\frac{N_{filters} \times T_{long}}{T_{tmp}}$ flows, and filtering gain

$$G = \frac{T_{long}}{T_{tmp}}$$

For example, assume $T_{tmp} = 1$ sec and $T_{long} = 10$ min. With 10,000 filters, V_{gw} blocks 10,000 flows/sec and keeps 6,000,000 flows blocked simultaneously.

If certain attack gateways and/or attack sources do not cooperate, undesired flows occur more than once. Suppose each undesired flow occurs on average n times. In this case, V_{gw} spends n temporary filters to cause 1 flow

to be blocked for T_{long} minutes. Thus, filtering rate, capacity and gain drop by a factor of n . For example, if all attack sources lie to their gateways, each undesired flow occurs twice. Then, with 10,000 filters, V_{gw} blocks 5,000 flows/sec and keeps 3,000,000 flows blocked simultaneously.

Victim's gateway — worst-case scenario: Now assume that none of the attack gateways cooperates, filter utilization exceeds its threshold, V_{gw} escalates all filtering requests, and all escalation attempts fail. In this situation, V_{gw} blocks traffic from all attack gateways to the victim locally, which means that with 1 filter, V_{gw} keeps 1 flow blocked. I.e., V_{gw} achieves filtering gain $G = 1$.

So, the maximum number of filters ever needed on V_{gw} to protect a single victim equals the total number of potential attack gateways. Note that this holds even if all undesired flows are spoofed — V_{gw} ends up *accepting* one flow from each alleged attack gateway, so it still needs as many filters as there can be attack gateways.

Surprisingly, the number of potential attack gateways is only a few tens of thousands. According to BGP data from Route Views [3] (retrieved in February 2005), there are currently 19,230 Autonomous Systems (ASes); of these, roughly 90% correspond to edge domains, while the rest are Internet Service Providers (ISPs). We processed the data with Gao's algorithm for inferring AS relationships [15] to get the number of providers per edge domain. Assuming a separate border router per customer-provider pair, there are only tens of thousands of border routers that could act as attack gateways.

To summarize, there may be millions of attack sources, but there are only tens of thousands of edge domains to host them. A router cannot accommodate a million filters, but it does accommodate tens of thousands. So, if eBay is under attack, eBay's gateway may be unable to locally block each attack source individually, but it *is* able to locally block each edge domain individually — i.e., each edge domain that does not cooperate to block its own misbehaving clients.

Attack gateway: Every time A_{gw} completes a handshake, it spends 1 filter for T_{tmp} seconds and causes 1 flow to be blocked. If the attack source complies (to avoid disconnection), the flow remains blocked for T_{long} minutes, as requested. So, with $N_{filters}$ filters, A_{gw} blocks $\frac{N_{filters} \times T_{long}}{T_{tmp}}$ flows, i.e., A_{gw} achieves filtering gain $G = \frac{T_{long}}{T_{tmp}}$. For example, assume $T_{tmp} = 1$ sec and $T_{long} = 10$ min. Suppose A_{gw} provides connectivity to 64,000 hosts (a class B network). With 256,000 filters, A_{gw} blocks 2,400 flows from each one of its clients.

If an attack source does not comply, the corresponding flow recurs before T_{long} minutes, A_{gw} completes a second handshake and spends again a temporary filter for T_{tmp} seconds. However, the attack source gets discon-

nected, which means that it does not come back online, unless it has been cleaned and patched. Thus, if attack sources do not comply, the filtering gain of the attack gateway actually increases, because infected hosts get disconnected, and A_{gw} does not have to filter their traffic again.

5.3 DRAM Requirements

Shadow memory: Both V_{gw} and A_{gw} keep a shadow filter for each flow that has been blocked. So, the maximum number of used shadow filters equals the maximum number of flows simultaneously blocked (i.e., the filtering capacity). Each shadow entry has form $\{A_{gw} : R \dots V_{gw} V\}$. Assuming R is 64 bits long (see Section 5.4) and flows are classified/filtered based on the last 6 components of their path, each shadow entry is 320 bits wide (2×32 bits for the IP source and destination, 6×32 bits for the non-random RR path components, and 64 bits for R). So, to keep a million flows blocked, V_{gw} needs about 40 MB of DRAM.

Note that DRAM is not the resource bottleneck; if it were not for the limited number of wire-speed filters, 40 GB of shadow memory would be enough to keep 1 billion undesired flows blocked.

Long-term filters on attack source: An attack source is not necessarily a malicious or compromised node; it is simply the sender of a traffic flow deemed undesired by its recipient. An "innocent" host classified as an attack source needs long-term filters to remember which receivers do not want its traffic (and avoid disconnection). Specifically, to satisfy n requests/sec by its provider, the host needs $n \times T_{long}$ filters. However, as opposed to wire-speed filters on routers, software filters on end-hosts are not a scarce resource.

Note that A does not risk disconnection for not satisfying requests beyond the agreed rate — a correctly functioning provider does not overload a customer with filtering requests and then disconnect the customer for failing to satisfy them.

5.4 Probability of Abuse

One potential attack against AITF is to try to guess the randomized RR header. For example, consider the topology of Figure 3 and suppose a set of malicious nodes (like M) spoof Stanford addresses and send high-rate traffic to eBay. V_{gw} contacts A_{gw} , gets the random value recorded by A_{gw} on all packets to eBay, and blocks all spoofed traffic. However, by sending a sufficiently large number of messages to eBay, the malicious nodes can try to guess the correct random value by brute force. If they succeed, they can successfully pretend to be Stanford hosts and potentially cause all traffic from Stanford to eBay to be classified as undesired and, thus, blocked.

To limit the probability of such abuse, A_{gw} changes the process by which it computes its random values every T_{change} minutes. If the random value is N bits long, to guess it with probability p , the malicious nodes must send $p \times 2^N$ messages. The amount of time it takes to send that many messages to eBay is bounded from below by eBay's maximum packet reception rate B . So, if the random value is N bits long, the malicious nodes can guess it during one T_{change} interval with probability $p = \frac{T_{change} \times B}{2^N}$. The probability that the malicious nodes guess one random value in T_{guess} minutes is the probability that they guess the random value in any of the $\frac{T_{guess}}{T_{change}}$ intervals:

$$P_{guess} = 1 - \left(1 - \frac{T_{change} \times B}{2^N}\right)^{\frac{T_{guess}}{T_{change}}}$$

For example, if eBay is connected through a 1 Gbps link, it receives up to 1.95 million packets/sec.¹ Suppose $T_{change} = 10$ minutes and $N = 64$ bits. The probability that the malicious nodes guess one random value in a month is 2.74×10^{-7} .

Changing the process that computes the random value creates the following problem: Suppose A_{gw} has communicated random values to a number of victim gateways; the moment it changes the process, all communicated random values become invalid. To avoid this problem, when A_{gw} responds to a handshake, it communicates to the victim's gateway, not only its current random value, but also the next $\frac{T_{long}}{T_{change}}$ random values and when they will be valid. By choosing $T_{change} = T_{long}$, A_{gw} must pre-compute one random value.

We should note that the size of the random value cannot be chosen solely based on the desired probability of abuse; it also affects the bandwidth overhead introduced by RR (because each RR-enabled border router adds a random value to each forwarded packet). We discuss RR bandwidth overhead in the Appendix, Section A.4.

Another potential attack against AITF is to try to compromise the 3-way handshake. For example, consider the topology of Figure 1 and suppose a set of malicious nodes pretend to be V 's gateway and initiate 3-way handshakes with A_{gw} , asking it to block all traffic to V . Assuming the malicious nodes are not on the path from A_{gw} to V , they do not see A_{gw} 's responses and the included nonce, necessary to complete the handshake. However, by sending a sufficiently large number of messages, they can guess the correct nonce by brute force. Following similar rational as above, the probability to guess one nonce in T_{guess} minutes depends on A_{gw} 's maximum packet reception rate. For example, if A_{gw} is connected through a 10 Gbps link, it receives up to 19.5 million packets/sec. Suppose $T_{change} = 10$ minutes and the

nonce size is 64 bits. The probability that the malicious nodes guess one nonce in a month is 2.74×10^{-6} . For a 128-bit nonce, the probability becomes 1.3×10^{-25} . Note that, unlike the random value, the nonce is not included in every packet forwarded by A_{gw} ; hence, the incentive to keep it small is less relevant.

5.5 Good Traffic Lost to Escalation

Escalation blocks all traffic from a non-cooperative attack gateway A_{gw} to the victim; clearly, this can lead to loss of good traffic. The decision to escalate or not is made by the victim, because the victim is the only one who can quantify the value of A_{gw} 's good traffic versus the damage caused by A_{gw} 's attack traffic. For example, suppose eBay is under attack by a million attack sources, all connected through AOL; at the same time, it is serving 1,000 legitimate AOL clients. If the AOL gateway does not cooperate, only eBay can decide whether serving the 1,000 good AOL clients is worth sustaining the 1,000,000 bad ones.

Whether a flow is "escalatable" or not depends on the policy module. Hence, we cannot compute a general estimate of the percentage of good traffic lost to escalation. However, we do implement a simple policy module in our simulation, and show how its decisions affect the victim's good traffic.

6 Simulation Results

We use real Internet routing table data to build a realistic simulation topology. We simulate different attack scenarios, where multiple attack sources (up to a million) attack a single victim, connected through a 100 Mbps link; the victim's gateway uses up to 10,000 filters to protect the victim. For each scenario, we plot the bandwidth of the attack traffic that reaches the victim as well as the victim's goodput as a function of time, i.e., we show how fast attack traffic is blocked and how much of the victim's goodput is restored.

6.1 Framework

We built our simulator within the Dartmouth Scalable Simulation Framework (DaSSF) [2]. To create our topology, we downloaded Internet routing table data from the Route Views project site [3]. We map each AS and each edge network to a separate AITF domain — we derive AS topology and peering relationships by applying Gao's algorithm for inferring inter-AS relationships [15] to the Route Views data; we derive edge network topology by roughly creating one edge network per advertised class A and class B prefix. Each AITF domain is represented by one AITF router. AITF routers are interconnected through OC-192 (9.953 Gbps) and OC-48 (2.488 Gbps) full-duplex links. End-hosts are connected to their

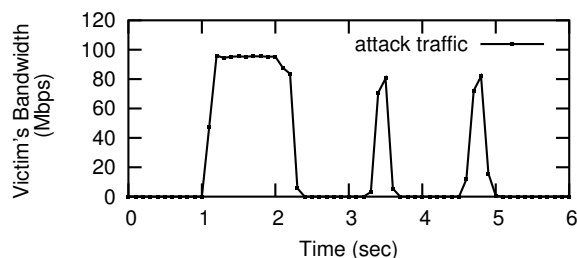


Figure 5: $t = 1$ sec: attack starts; $t = 2 - 3$ sec: V detects the attack and sends 10,000 filtering requests to its gateway; $t = 3 - 4$ sec: flows recur for the first time; $t = 4 - 5$ sec: flows recur for the second time, and V_{gw} blocks all traffic from the compromised gateways.

routers through Fast (100 Mbps) and Thin (10 Mbps) Ethernet full-duplex links. Internet round-trip times average 200 msec. Host-to-router round-trip times average 20 msec. In all scenarios, $T_{tmp} = 1$ sec and $T_{long} = 2$ min.

6.2 Filtering Response Time

Our first experiment demonstrates that AITF achieves a filtering response time equal to the one-way delay from the victim to its gateway, i.e., on the order of milliseconds. It also demonstrates that “lying” gateways are quickly detected and blocked; the worst each lying gateway can do is cause up to two “spikes” spaced out by at least T_{tmp} seconds.

Scenario 1: The victim receives a flooding attack by 10,000 attack sources, each behind its own attack gateway. The bandwidth of the attack (before defense) is 1 Gbps. All attack gateways are lying, i.e., they agree to block their undesired flows and then break the agreement.

Figure 5 illustrates that V_{gw} blocks attack traffic within milliseconds from the moment the attack is detected; attack traffic recurs twice and is completely blocked within 2 seconds. V_{gw} gives two chances to each attack gateway to honor its filtering agreement; when the attack gateways break their agreements twice, all their traffic to V is blocked. We run the experiment only for 10,000 undesired flows (which allows the victim to have all of them blocked within 1 sec), so that the “spike” effect due to the recurring flows is visible.

6.3 Filtering Gain

The next two experiments demonstrate that the victim’s gateway can achieve filtering gain on the order of hundreds, i.e., the victim’s gateway blocks two orders of magnitude more flows than it uses filters.

Scenario 2: The victim receives a flooding attack by 100,000 attack sources. The bandwidth of the attack

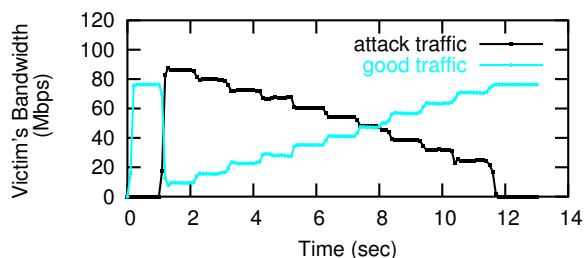


Figure 6: $t = 0 - 1$ sec: V is receiving ~ 80 Mbps of goodput; $t = 1 - 2$ sec: attack drops V ’s goodput to 12% of original; $t = 2$ sec: V starts sending 10,000 filtering requests/sec to its gateway; $t = 12$ sec: V ’s goodput is restored to 100% of original.

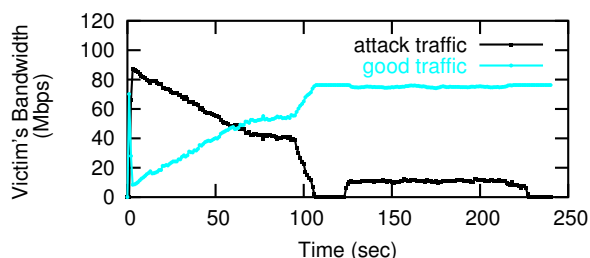


Figure 7: $t = 0 - 1$ sec: V is receiving ~ 80 Mbps of goodput; $t = 1 - 2$ sec: attack drives V ’s goodput to 12% of original; $t = 2$ sec: V starts sending 10,000 filtering requests/sec to its gateway; $t = 104$ sec: V ’s goodput is restored to 100% of original; $t = 122$ sec: filtering requests start expiring, undesired flows are released and re-blocked, 10,000 at a time.

(before defense) is 1 Gbps. The victim’s goodput (before the attack) is approximately 80 Mbps. All attack gateways cooperate.

Scenario 3: Similar to scenario 2, but the victim receives a flooding attack by 1,000,000 attack sources.

Figures 6 and 7 show that, using 10,000 filters, V_{gw} blocks 100,000 flows in 10 seconds and 1,000,000 flows in 100 seconds. Without AITF, a router needs a million filters to block a million flows; these experiments demonstrate that, with AITF, V_{gw} needs only ten thousand filters to block a million flows. Hence, AITF reduces the number of filters required to block a certain number of flows by two orders of magnitude — a critical improvement, since routers typically accommodate tens of thousands of filters, whereas DDoS attacks can easily consist of millions of flows. Figure 7 also reveals what happens after $T_{long} = 2$ minutes. We assume that attack sources are “smart”, i.e., they pause sending an undesired flow when so requested (to avoid disconnection) and they restart after $T_{long} = 2$ minutes.

6.4 Escalation and Lost Goodput

The last two experiments illustrate the trade-off involved in policy decisions regarding non-cooperative gateways. The victim’s gateway may decide to escalate and lose

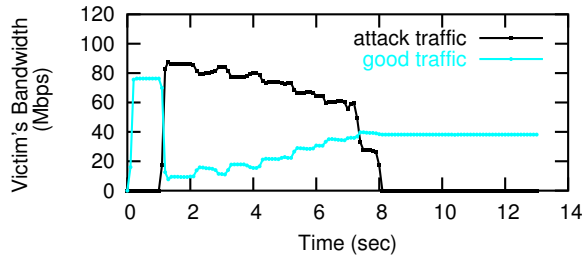


Figure 8: $t = 0 - 1$ sec: V is receiving ~ 80 Mbps of goodput; $t = 1 - 2$ sec: attack drives V 's goodput to 12% of original; $t = 2$ sec: V starts sending 10,000 filtering requests/sec to its gateway; $t = 2 - 8$ sec: half of the attack gateways do not cooperate, so V_{gw} escalates and blocks all their traffic; $t = 8$ sec: V 's goodput is restored to 50% of original.

all traffic from a non-cooperative gateway to the victim; alternatively, it may decide to locally block as many undesired flows as possible and let the rest through. Although we did not implement a complete policy module, we choose two “extreme” scenarios, each favoring a different decision, implement the best policy for each scenario, and show the results.

Scenario 4: The victim receives a flooding attack, but, this time, half of the attack gateways are non-cooperative. Good and bad sources are collocated, i.e., evenly distributed behind the same gateways. The attack comes from 100,000 attack sources; attack bandwidth (before defense) is 1 Gbps. The victim's goodput (before the attack) is approximately 80 Mbps.

Figure 8 shows that, using 10,000 filters, V_{gw} restores 50% of the victim's goodput in 6 seconds. In this scenario, attack traffic has 10 times the rate of good traffic. The policy module chooses to block all traffic from non-cooperative attack gateways. This cannot make things any worse, since most good traffic is being dropped anyway due to the flood; on the contrary, it allows good traffic from cooperative gateways to get through.

Scenario 5: In this scenario, all attack gateways are non-cooperative, and good and bad sources are collocated. However, the attack comes from fewer attack sources (20,000) and consumes lower bandwidth (160 Mbps, before defense). The victim's goodput (before the attack) is approximately 80 Mbps.

Figure 9 shows that, using 10,000 filters, V_{gw} restores 75% of the victim's goodput in a few milliseconds from the moment the attack is detected. This scenario is trickier than the previous one, because attack traffic has only twice the rate of good traffic. In this case, escalating would be disastrous — it would drop goodput to 0, because good and bad sources are collocated. The policy module chooses to block as many flows as possible locally and let the rest through. This results in half the attack traffic being dropped and half getting through,

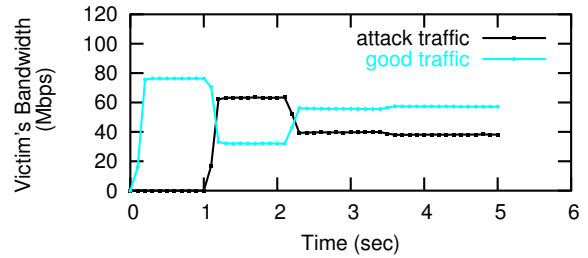


Figure 9: $t = 0 - 1$ sec: V is receiving ~ 80 Mbps of goodput; $t = 1 - 2$ sec: attack drives V 's goodput to $\sim 50\%$ of original; $t = 2$ sec: V starts sending 10,000 filtering requests/sec to its gateway; none of the attack gateways cooperate, so V_{gw} blocks locally as many flows as possible and lets the rest through; $t = 2 - 3$ sec: V 's goodput is restored to $\sim 75\%$ of original.

which allows only 75% of the victim's goodput to get through. I.e., the victim prefers to sustain some attack traffic from non-cooperative attack gateways, rather than sacrificing all their good traffic.

7 Deployment

This section describes how AITF can be deployed in today's Internet. We also consider the incentives for early adopters and compatibility with legacy hosts.

7.1 Model

Rather than requiring every Internet router to support AITF, it is sufficient for the border routers between administrative domains to support it. We introduce the notion of an *AITF domain* as an administrative domain whose border routers are AITF-enabled.

An AITF domain has a *filtering contract* with each local end-host and peering domain. Such a contract specifies a maximum filtering request rate, i.e., the maximum rate at which the AITF domain can send/receive requests to block undesired flows to/from each end-host and peering domain. An AITF domain enforces the specified rates and indiscriminately drops messages from an end-host/domain when that party exceeds the agreed rate.

In a way, an AITF domain is the “dual” of a BGP Autonomous System (AS): ASes exchange routing information, which communicates their willingness to relay certain packets. Similarly, AITF domains exchange filtering information, which communicates their *unwillingness* to receive certain packets. However, an AITF domain differs from an AS, in that it exchanges messages with other AITF domains that are not adjacent to it — recall that the victim's gateway talks directly to the attack gateway. It seems natural for every AS to map to a separate AITF domain. Popular public-access sites, like eBay or Amazon, may even deploy AITF on their corporate firewalls, to avoid sharing a victim's gateway with other customers of their AS.

Our position is that the filtering contract should be part of the Service Level Agreement (SLA) signed between the customer and the service provider. In this way, when a domain agrees to provide a certain amount of bandwidth to a customer, the provider also agrees to satisfy a certain rate of filtering requests coming from that customer. At the same time, the customer agrees to satisfy a certain rate of filtering requests coming from the provider. The customer-provider pair can be an end-host and an edge network, or an edge network and an ISP, or even a small ISP and a backbone network.

7.2 Incentive for Initial Deployment

It makes sense for AITF deployment to begin at the edges: An edge network that hosts potential DDoS victims (e.g., a web hosting domain) deploys AITF to protect its clients; an edge network that hosts potential attack sources (e.g., a campus network or a dialup provider) deploys AITF to maintain its connectivity to public-access sites, even when these sites are under attack.

For example, in Figure 3, we depicted an early deployment stage, where only eBay and Stanford have deployed AITF. Suppose a worm compromises millions of attack sources, uniformly distributed across the Internet, and commands them to send high-rate traffic to eBay. Without AITF, there is nothing eBay can do; almost all its good traffic is lost in the flood. With AITF, eBay identifies the undesired flows coming from Stanford; eBay's gateway exchanges a handshake with Stanford's gateway, which agrees to block its undesired flows to eBay; eBay's gateway accepts all traffic from Stanford and drops (or rate-limits) all the rest — most good traffic to eBay is lost anyway due to the attack, so dropping traffic from AITF-unaware domains cannot make things worse. I.e., Stanford is the only domain that cooperates with eBay and, hence, the only domain to maintain its connectivity to eBay throughout the attack.

To conclude, the first domains to deploy AITF benefit, because they preserve their connectivity to each other in the face of DDoS attacks. As AITF deployment spreads, the benefit of AITF-enabled domains grows, because they maintain their connectivity to larger and larger portions of the Internet.

7.3 Compatibility with Legacy Hosts

AITF involves a rather draconian measure against attack sources: Either they stop sending an undesired flow or they get disconnected. One could argue that malicious node M can abuse this measure to disconnect legacy host L : M sends a filtering request to L 's gateway to block all traffic from L to M ; L 's gateway sends a similar request to L (which is ignored, since L is AITF-unaware); then M tricks L into sending it traffic (e.g., sends an ICMP request); as a result, L 's gateway disconnects L .

This scenario cannot happen. Recall that an AITF domain has a filtering contract with each of its customers; the contract specifies the maximum rate at which the domain sends filtering requests to the customer. An AITF-unaware customer by definition has agreed to rate 0; hence, its provider never sends filtering requests to it.

A domain that deploys AITF either forces its end-hosts to deploy AITF as well, or accepts the cost of filtering their undesired flows; for example, L 's gateway locally blocks all traffic from L to M for T_{long} minutes. The second option is more incrementally deployable, but requires more filtering resources from an attack gateway — namely, as many filters as undesired flows generated by its end-hosts. As a compromise, a provider can charge legacy customers that do not support AITF, for the potential cost induced by their inability to block their undesired traffic themselves.

8 Discussion of Additional Attacks

Malicious nodes collocated with the victim: The handshake between the victim's gateway and the attack gateway protects only the communication between these two entities; it does not protect the communication between the victim and the victim's gateway. A malicious end-host located on the same LAN with end-host V can clearly spoof V 's address, send filtering requests as V , and disrupt V 's communications.

An AITF domain can easily avoid such abuses by either preventing source address spoofing in its own network or authenticating local filtering requests. Note that the latter does not require any public key infrastructure; it only requires from each border router of an AITF domain to share a secret with each of the domain's customers.

DDoS against AITF: One could argue that a set of malicious nodes can launch the following attack against router A_{gw} : First, pretend they are victim gateways and send a high rate of filtering requests to exhaust A_{gw} 's filters. Once A_{gw} 's filters are exhausted, a malicious node located behind A_{gw} is commanded to start an undesired flow against, say, Google; Google asks from A_{gw} to stop, A_{gw} has no filters left and gets disconnected from Google.

The first thing to note is that the attack gateway uses the 3-way handshake to verify the “authenticity” of a filtering request, i.e., that the requester is on the path to the alleged victim. Once a request is deemed authentic, the identity of the requesting victim gateway is established, and the attack gateway can accept or drop the corresponding request based on that. The attack gateway satisfies up to a certain rate of requests from each victim gateway to avoid exhausting all its filters to satisfy a few demanding domains.

Now consider a set of malicious nodes seeking to attack router A_{gw} . One way is to send to A_{gw} a high rate of authentic filtering requests; the only thing they accomplish is to exhaust the quota of their own domains.² The other way is to send to A_{gw} a high rate of spoofed filtering requests; now their requests are dropped. The only harm the malicious nodes can do is launch a SYN-flood-style attack, i.e., flood A_{gw} with fake requests, hoping to exhaust the processing cycles devoted to 3-way handshakes.

There are two steps to dealing with such attacks. The first one is to implement the part of the attack gateway algorithm that deals with the 3-way handshake in the fast path — it just involves hashing certain contents of a received message and comparing the result to a value included in the message. The second step is to let A_{gw} act as the victim: if a set of malicious nodes manage to flood A_{gw} 's AITF (or any other) hardware module, then A_{gw} uses the RR headers of incoming traffic to identify undesired flows and asks from its own gateway to have them blocked.

Malicious on-the-path nodes: The handshake between the victim's gateway and the attack gateway prevents malicious node M from installing a filter on router A_{gw} to block certain traffic to router V_{gw} , as long as M is off the path from A_{gw} to V_{gw} . A malicious node on the path from A_{gw} to V_{gw} can clearly forge filtering requests and disrupt A_{gw} - V_{gw} communication.

However, a malicious node on the path from A_{gw} to V_{gw} can only be an Internet core router; such a malicious router can disrupt A_{gw} - V_{gw} communication anyway, e.g., by blocking all their traffic. I.e., if a core router gets compromised, thousands of domains that connect through that router are at its mercy — it makes little difference whether the router is AITF-enabled or not.

9 Related Work

Packet Marking: The alternative to route record is probabilistic packet marking (PPM)[21, 14]: Each participating router marks each forwarded packet with certain probability p ; a DDoS victim combines marks from multiple packets to identify the routers that forward attack traffic. Instead of using a separate header, PPM uses a lightly utilized IP header field (typically the 16-bit IP identifier); doing so facilitates deployment and decreases packet overhead. Although a promising and incrementally deployable traceback technique, PPM is less useful in actually blocking attack traffic. Identifying the routers that forward attack traffic is not enough; the victim's gateway V_{gw} must identify *their traffic* in order to block it. Even if attack gateway A_{gw} agrees to block an undesired flow, V_{gw} must still identify A_{gw} 's traffic in order to verify that A_{gw} is honoring the agreement. Therefore,

V_{gw} must be able to identify the path followed by each incoming packet at wire speed. The only way to perform path-based wire-speed filtering is to have each packet's path deterministically recorded on the packet.

One could certainly argue for “compressing” the path — no need to record one full 32-bit address per border router! In Path Identifier (Pi) [23], each participating router deterministically marks the forwarded packets, so that each packet obtains a “fingerprint” that reflects the entire path followed by the packet. Indeed, this approach enables the victim (or its gateway) to locally block undesired flows in the face of source address spoofing. However, it does not meet the two following requirements: (i) V_{gw} must know the addresses of the border routers on an undesired flow's path; otherwise, it must locally block all attack flows itself, which is typically beyond its capabilities. (ii) V_{gw} must be able to block attack traffic at different granularities, e.g., block all packets with path $\{* A_{gw} \dots V_{gw} V\}$.

Filtering: The Pushback scheme [18] uses hop-by-hop filter propagation to push filtering of undesired traffic away from the victim: The victim identifies the upstream routers that forward attack traffic to it and sends them filtering requests; the routers satisfy the requests, potentially identify the next upstream routers that forward attack traffic to the victim, and send them similar requests. Each Pushback router installs a single filter per victim and rate-limits all traffic to each victim. The main benefit of this approach is that it does not require knowledge of the attack paths, which makes it deployable without packet marking. However, when attack traffic and good traffic share common paths, and attack traffic is of much higher rate than good traffic (increasingly the case, nowadays), rate-limiting all traffic to the victim sacrifices most of the victim's good traffic.

The Stateless Internet Flow Filter (SIFF) [9] divides all Internet traffic into privileged and non-privileged. A client establishes a privileged channel to a server through a capability exchange handshake; the client includes the capability in each subsequent packet it sends to the server; each router along the path verifies the capability and gives priority to privileged traffic. The main advantage of SIFF is that it does not require any filtering state in the routers. However, once a server is under attack, a new client must contend with attack traffic to establish a connection — because channel establishment involves an exchange of non-privileged packets. It also requires counter-measures to prevent malicious nodes from establishing privileged channels between themselves and flood the network with privileged traffic.

Secure Overlays: The set of AITF-enabled border routers can be viewed as a “filtering overlay”. Filtering overlays have also been suggested as a way to prevent

DoS against critical applications (like Emergency Services) that are meant to be accessed only by authorized users [16, 12]. In that context, the overlay nodes perform client authentication and relay traffic to a protected server, whose IP address is unknown outside the overlay.

10 Conclusion

We presented Active Internet Traffic Filtering (AITF), a mechanism for filtering highly distributed denial-of-service attacks. We showed that AITF can block a million undesired flows, while requiring only tens of thousands of wire-speed filters from each participating router — an amount easily accommodated by today’s routers. It also prevents abuse by malicious nodes seeking to disrupt other nodes’ communications.

More specifically, we showed the following:

1. AITF offers filtering response time equal to the one-way delay from the victim to the victim’s gateway. I.e., a victim can have an undesired flow blocked within milliseconds.

2. AITF offers filtering gain on the order of hundreds of blocked flows per used filter. I.e., a router can block two orders of magnitude more flows than it has wire-speed filters. For example, suppose eBay is receiving a million undesired flows; with 10,000 filters, eBay’s gateway can have all flows blocked within 100 seconds. In the worst-case scenario, eBay’s gateway blocks all traffic from each domain that hosts attack sources and refuses to filter their traffic, which (in today’s Internet) requires a few tens of thousands of filters.

3. A set of malicious nodes can practically not abuse AITF to disrupt communication from node *A* to node *B*, as long as they are not located on the path from *A* to *B*. This holds even during initial deployment, where most Internet domains are AITF-unaware.

The idea behind AITF is that the Internet *does* have enough filtering capacity to block large amounts of undesired flows — it is just that this capacity is concentrated close to the attack sources. AITF enables service providers to “gain access” to this filtering capacity and couple it with a reasonable amount of their own filtering resources, in order to protect their customers in the face of increasingly distributed denial-of-service attacks.

11 Acknowledgments

We would like to thank Daniel Faria, Evan Greenberg, Dapeng Zhu, George Candea, Nagendra Modadugu, Costa Sapuntzakis, Tassos Argyros and our shepherd, Mema Roussopoulos, for their constructive feedback.

A Appendix: Route Record

A.1 Header Update

The RR header consists of three fields: the *path* is a list of (initially empty) slots; the *size* is the total number of slots in the path; the *pointer* points to the first empty slot in the path. The first RR-enabled border router on a packet’s path (i) inserts an RR header in the packet; (ii) writes its own IP address and random value in the first slot; and (iii) sets the pointer to point to the second slot. Each subsequent RR-enabled border router that ingresses the packet into a new AS (i) reads the pointer; (ii) writes its own IP address and random value in the indicated slot; and (iii) increments the pointer to point to the next slot. If there is no room left in the RR header, the router drops the packet.

A.2 Hardware Support

Each RR-enabled border router updates the RR headers of forwarded packets as described in Section A.1. RR-header update requires (i) computing a keyed hash function on the packet’s destination, (ii) reading and modifying the RR pointer, and (iii) writing the router’s address and random value on the indicated slot. So, although the RR header has a variable length, its update requires reading/modifying a 4-bit pointer and a 96-bit path component (assuming 64-bit random values). Computing a 64-bit hash (like HMAC-SHA1) per forwarded packet can be easily done today at wire-speed [11].

As mentioned in Section A.1, RR headers are not inserted by end-hosts; they are inserted by the first border router on each packet’s path — call it the packet’s “gateway”. So, when an edge network deploys RR, it upgrades its border routers to support RR-header insertion. A border router determines the required RR header size for each packet, by looking up the AS path length to the packet’s destination domain as communicated through BGP. If the real AS path turns out to be longer than the communicated AS path, the packet gets dropped; the packet’s gateway receives an ICMP message, increases the RR header size and retransmits the packet — i.e., AS path length discovery is similar to MTU discovery using the “Don’t Fragment” IP-header flag. Once a router discovers the real AS path length to a destination domain, it caches its value to avoid future retransmissions.

RR-header insertion is similar to packet encapsulation, a well-studied operation, for which multiple hardware implementations already exist. Note that, as stated in Section 7.2, RR/AITF deployment starts at the edges; i.e., it is edge routers that insert RR headers, not core routers connected to Internet backbones. Edge routers are connected at best through OC-48 (2.488 Gbps) links; the Cisco 10000 Series OC-48c linecard already supports encapsulation.

A.3 Compatibility with Legacy Hosts

Before inserting an RR header in an outgoing packet, the packet's gateway must make sure that the receiving domain has deployed RR; otherwise, the receiver will not recognize the RR header and drop the packet. Hence, each packet gateway keeps track of the destination ASes to which it forwards packets, and asks them whether they care to receive RR headers.

To avoid introducing latency, a packet gateway forwards packets without inserting RR headers and starts doing so as soon as it concludes that the destination domain is RR-enabled. To avoid querying the destination domain on each packet, it periodically queries potential destination domains (e.g., once per hour) and caches their response. Note that the number of potential destination domains can be no bigger than the number of Internet ASes — 19,230 for the current Internet.

A.4 Bandwidth Overhead

RR bandwidth overhead depends on the average number of ASes per packet path. Although we do not know this number, we can approximate it with the average number of AS-level hops between Internet ASes, which is close to 4 [17, 10]. Assuming an average of 4 ASes per Internet path and 64-bit random values, route record introduces on average 49 extra bytes per packet. For an average packet size of 500 bytes [5], this leads to 10% bandwidth overhead.

Bandwidth overhead can be reduced to 5% by using the following twist: Instead of each router adding a random value to the recorded path, only one router does so; the first border router that forwards the packet into an RR-unaware domain. For example, consider the topology in Figure 1; suppose only A_{NET} , A_{ISP} and V_{NET} have deployed AITF. If a packet is sent from A to V , the only router that adds a random value to the packet's header is X . This reduces bandwidth overhead, at the cost of restricting the routing of filtering requests: If V sends a filtering request against A 's traffic, that request must be routed through X , so that X verifies the authenticity of the recorded path.

Notes

¹To make our analysis conservative, we assume that eBay responds to all the messages sent by the malicious nodes. In reality, if a set of nodes send such high-rate traffic to eBay, eBay will consider their traffic undesired and use AITF to have it blocked.

²An AITF-enabled border router can prevent its own clients from pretending to be victim gateways and exhausting its quota, simply by dropping all outgoing filtering requests — recall that no other node but the border router itself is supposed to send filtering requests outside the local domain.

References

- [1] Access list configuration in Cisco's Gigabit Ethernet Interface. http://www.cisco.com/en/US/products/hw/switches/ps5304/prod_configuration_guides_list.html.
- [2] Dartmouth scalable simulation framework. <http://www.crhc.uiuc.edu/~jasonliu/projects/ssf/>.
- [3] Route Views Archive. <http://archive.routeviews.org/oix-route-views/>.
- [4] SiberCore SCT1842 Features. http://www.sibercore.com/products_siberCAM.htm#3.
- [5] Sprint backbone data. <http://ipmon.sprint.com/packstat/packetoverview.php>.
- [6] SCO Offline from Denial-of-Service Attack. <http://www.caida.org/analysis/security/sco-dos/>, December 2003.
- [7] Attack downs Yahoo, Google. <http://news.zdnet.co.uk/internet/security/0,39020375,39157748,00.htm>, June 2004.
- [8] DDoS Attack Knocks Out DoubleClick Ads. <http://www.eweek.com/article2/0,1759,1628340,00.asp>, July 2004.
- [9] SIFF: A Stateless Internet Flow Filter to Mitigate DDoS Flooding Attacks. In *IEEE Security and Privacy*, May 2004.
- [10] BGP table data. <http://bgp.potaroo.net/index-bgp.html>, February 2005.
- [11] Personal communication with Fusun Ertemalp, Distinguished Engineer at Cisco Systems, February 2005.
- [12] D. G. Andersen. Mayday: Distributed Filtering for Internet Services. In *USITS*, March 2003.
- [13] D. J. Bernstein. SYN Cookies. <http://cr.yp.to/syncookies.html>, 1996.
- [14] D. Dean, M. Franklin, and A. Stubblefield. An Algebraic Approach to IP Traceback. *NDSS*, February 2001.
- [15] L. Gao. On Inferring Autonomous System Relationships in the Internet. In *Global Internet*, November 2000.
- [16] A. D. Keromytis, V. Misra, and D. Rubenstein. Secure Overlay Services. In *ACM SIGCOMM*, August 2002.
- [17] D. Magoni and J. J. Pansiot. Analysis of the Autonomous System Network Topology. *ACM CCR*, 31(3):26–37, July 2001.
- [18] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Schenker. Controlling High Bandwidth Aggregates in the Network. *ACM CCR*, 32(3):62–73, July 2002.
- [19] D. A. Patterson. A simple way to estimate the cost of downtime. In *USENIX Systems Administration Conference*, November 2002.
- [20] J. Postel. RFC 791 - Internet Protocol.
- [21] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *ACM SIGCOMM*, August 2000.
- [22] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *USENIX Security*, August 2002.
- [23] A. Yaar, A. Perrig, and D. Song. Pi: A Path Identification Mechanism to Defend against DDoS Attacks. In *IEEE Security and Privacy*, May 2003.

Building a Reactive Immune System for Software Services

Stelios Sidiroglou Michael E. Locasto Stephen W. Boyd Angelos D. Keromytis
Network Security Lab
Department of Computer Science, Columbia University
{*stelios, locasto, swb48, angelos*}@cs.columbia.edu

Abstract

We propose a *reactive* approach for handling a wide variety of software failures, ranging from remotely exploitable vulnerabilities to more mundane bugs that cause abnormal program termination (*e.g.*, illegal memory dereference) or other recognizable bad behavior (*e.g.*, computational denial of service). Our emphasis is in creating “self-healing” software that can protect itself against a recurring fault until a more comprehensive fix is applied.

Briefly, our system monitors an application during its execution using a variety of external software probes, trying to localize (in terms of code regions) observed faults. In future runs of the application, the “faulty” region of code will be executed by an instruction-level emulator. The emulator will check for recurrences of previously seen faults before each instruction is executed. When a fault is detected, we recover program execution to a safe control flow. Using the emulator for small pieces of code, as directed by the observed failure, allows us to minimize the performance impact on the immunized application.

We discuss the overall system architecture and a prototype implementation for the *x86* platform. We show the effectiveness of our approach against a range of attacks and other software failures in real applications such as Apache, *sshd*, and Bind. Our preliminary performance evaluation shows that although full emulation can be prohibitively expensive, selective emulation can incur as little as 30% performance overhead relative to an uninstrumented (but failure-prone) instance of Apache. Although this overhead is significant, we believe our work is a promising first step in developing self-healing software.

1 Introduction

Despite considerable work in fault tolerance and reliability, software remains notoriously buggy and crash-prone. The situation is particularly troublesome with respect to services that must maintain high availability in the

face of remote attacks, high-volume events (such as fast-spreading worms like Slammer [2] and Blaster [1]) that may trigger unrelated and possibly non-exploitable bugs, or simple application-level denial of service attacks. The majority of solutions to this problem fall into four categories:

- **Proactive approaches** that seek to make the code as dependable as possible, through a combination of safe languages (*e.g.*, Java), libraries [3] and compilers [15], code analysis tools [8], and development methodologies.
- **Debugging aids** whose aim is to make post-fault analysis and recovery as easy as possible for the programmer.
- **Runtime solutions** that seek to contain the fault using some type of sandboxing, ranging from full-scale emulators such as VMWare, to system call sandboxes [24], to narrowly applicable schemes such as StackGuard [13].
- **Byzantine fault-tolerance schemes** (*e.g.*, [34]) which use voting among a number of service instances to select the correct answer, under the assumption that only a minority of the replicas will exhibit faulty behavior.

The contribution of this paper is a *reactive* approach, accomplished by observing an application (or appropriately instrumented instances of it) for previously unseen failures. The types of faults we focus in this paper consist of illegal memory dereferences, division by zero exceptions, and buffer overflow attacks. Other types of failures can be easily added to our system as long as their cause can be algorithmically determined (*i.e.*, another piece of code can tell us what the fault is and where it occurred). We intend to enrich this set of faults in the future; specifically, we plan to examine Time-Of-Check-To-Time-Of-Use (TOCTTOU) violations, and algorithmic-complexity denial of service attacks [9].

Our approach employs an Observe Orient Decide Act (OODA) feedback loop and uses a set of software probes that monitor the application for specific types of faults. Upon detection of a fault, we invoke a localized recovery mechanism that seeks to recognize and prevent the specific failure in future executions of the program. Using continuous hypothesis testing, we verify whether the fault has been repaired by re-running the application against the event sequence that apparently caused the failure. Our initial focus is on automatic healing of services against newly detected faults (whether accidental failures or attacks). We emphasize that we seek to address a wide variety of software failures, not just attacks.

For our recovery mechanism we introduce Selective Transactional EMulation (*STEM*), an instruction-level emulator that can be selectively invoked for arbitrary segments of code, allowing us to mix emulated and non-emulated execution inside the same process. The emulator allows us to (a) monitor for the specific type of failure prior to executing the instruction, (b) undo any memory changes made by the function inside which the fault occurred, by having the emulator record all memory modifications made during its execution, and (c) simulate an error-return from said function.

One of our key assumptions is that we can create a mapping between the set of errors that *could* occur during a program's execution and the limited set of errors that are explicitly handled by the program's code. This "error virtualization" technique is based on heuristics that we present in Section 2.4. We believe that a majority of server applications are written to have relatively robust error handling; by virtualizing the errors, an application can continue execution even though a boundary condition that was not predicted by the programmer allowed a fault to "slip in." In other words, error virtualization attempts to retrofit an exception catching mechanism onto code that wasn't explicitly written to have such a capability. Our experiments with Apache, OpenSSH, and Bind validate this intuition. Evidence from other recent work [26, 25, 33] supports our findings.

Our current work focuses on server-type applications, since they typically have higher availability requirements than user-oriented applications. Micro-rebooting [7] has been proposed as another approach to dealing with errors, by restarting all or parts of an application upon recognizing a failure. However, server applications often cannot be simply restarted because they are typically long running (and thus accumulate a fair amount of state) and usually contain a number of threads that service many remote users. Restarting the whole server because of one failed thread unfairly denies service to other users. Also, unlike user-oriented applications, servers operate

without direct human supervision and thus have a higher need for an automated reactive system. Furthermore, it is relatively easy to replay the offending sequence of events in such applications, as these are typically limited to input received over the network (as opposed to a user's interaction with a graphical interface). We intend to investigate other classes of applications in the future.

To evaluate the effectiveness of our system and its impact to performance, we conduct a series of experiments using a number of open-source server applications including Apache, *sshd*, and Bind. The results show that our "virtualized error" mapping assumption holds for more than 88% of the cases we examined. Testing with real attacks against Apache, OpenSSH, and Bind, we show that our technique can be effective in quickly and automatically protecting against zero-day attacks and failures. Although full emulation of these is prohibitively expensive (3,000% slowdown), our selective emulation degrades performance by a factor of 1.3–2, depending on the size of the emulated code segment. We believe that our findings show that a reactive approach such as we advocate is a promising mechanism for dealing with application faults.

Paper Organization. Section 2 presents our approach, including the limitations of our system and the basic system architecture. Section 3 briefly discusses the implementation of *STEM*, and Section 4 presents some preliminary performance measurements of the system. We give an overview of related work in Section 5 and summarize our contributions and plan for future work in Section 6.

2 Approach

Our architecture, depicted in Figure 1, uses three types of components: a set of sensors that monitor an application (such as a web server) for faults; Selective Transactional EMulation (*STEM*), an instruction-level emulator that can selectively emulate "slices" (arbitrary segments) of code; and a testing environment where hypotheses about the effect of various fixes are evaluated. These components can operate without human supervision to minimize reaction time.

2.1 System Overview

When the sensor detects an error in the application's execution (such as a segmentation fault), the system instruments the portion of the application's code that immediately surrounds the faulty instruction(s), such that the

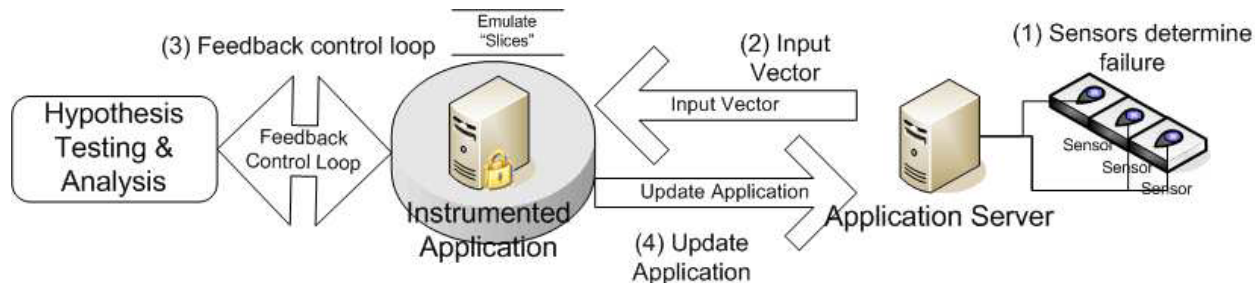


Figure 1: **Feedback control loop:** (1) a variety of sensors monitor the application for known types (but unknown instances) of faults; (2) upon recognizing a fault, we emulate the region of code where the fault occurred and test with the inputs seen before the fault occurred; (3) by varying the scope of emulation, we can determine the “narrowest” code slice we can emulate and still detect and recover from the fault; (4) we then update the production version of the server.

code segment is emulated (the mechanics of this are explained in Section 3). To verify the effectiveness of the fix, the application is restarted in a test environment with the instrumentation enabled, and is supplied with the input that caused the failure (or the N most recent inputs, if the offending one cannot be easily identified, where N is a configurable parameter). We focus on server type applications that have a transactional processing model, because it is easier to quickly correlate perceived failures with a small or finite set of inputs than with other types of applications (e.g., those with a GUI).

During emulation, *STEM* maintains a record of all memory changes (including global variables or library-internal state, e.g., *libc* standard I/O structures) that the emulated code makes, along with their original values. Furthermore, *STEM* examines the operands for each machine instruction and pre-determines the side effects of the instructions it emulates. The use of an emulator allows us to circumvent the complexity of code analysis, as we only need to focus on the operation and side effects of individual instructions independently from each other.

If the emulator determines that a fault is about to occur, the emulated execution is aborted. Specifically, all memory changes made by the emulated code are undone, and the currently executing function is “forced” to return an error. We describe how both emulation and error virtualization are accomplished in Sections 2.3 and 2.4, respectively, and we experimentally validate the error virtualization hypothesis in Section 4. For our initial approach, we are primarily concerned with failures where there is a one-to-one correspondence between inputs and failures, and not with those that are caused by a combination of inputs. Note, however, that many of the latter type of failures are in fact addressed by our system, because the last input (and code leading to a failure) will be recognized as “problematic” and handled as we have discussed.

In the testing and error localization phase, emulation

stops after forcing the function to return. If the program crashes, the scope of the emulation is expanded to include the parent (calling) routine and the application re-executes with the same inputs. This process is repeated until the application does not terminate after we abort a function calls sequence. In the extreme case, the whole application could end up being emulated, at a significant performance cost. However, Section 4 shows that this failsafe measure is rarely necessary.

If the program does not crash after the forced return, we have found a “vaccine” for the fault, which we can use on the production server. Naturally, if the fault is not triggered during an emulated execution, emulation halts at the end of the vulnerable code segment, and all memory changes become permanent.

The overhead of emulation is incurred at all times (whether the fault is triggered or not). To minimize this cost, we must identify the smallest piece of code that we need emulate in order to catch and recover from the fault. We currently treat functions as discrete entities and emulate the whole body of a function, even though the emulator allows us to start and stop emulation at arbitrary points, as described in Section 3. Future work will explore strategies for minimizing the scope of the emulation and balancing the tradeoff between coverage and performance.

In the remainder of this section, we describe the types of sensors we employ, give an overview of how the emulator operates (with more details on the implementation in Section 3), and describe how the emulator forces a function to return with an error code. We also discuss the limitations of reactive approaches in general and our system in particular.

2.2 Application Monitors

The selection of appropriate failure-detection sensors depends on both the nature of the flaws themselves and tolerance of their impact on system performance. We describe the two types of application monitors that we have experimented with.

The first approach is straightforward. The operating system forces a misbehaving application to abort and creates a core dump file that includes the type of failure and the stack trace when that failure occurred. This information is sufficient for our system to apply selective emulation, starting with the top-most function in the stack trace. Thus, we only need a watchdog process that waits until the service terminates before it invokes our system.

A second approach is to use an appropriately instrumented version of the application on a separate server as a honeypot, as we demonstrated for the case of network worms [29]. Under this scheme, we instrument the parts of the application that may be vulnerable to a particular class of attack (in this case, remotely exploitable buffer overflows) such that an attempt to exploit a new vulnerability exposes the attack vector and all pertinent information (attacked buffer, vulnerable function, stack trace, *etc.*).

This information is then used to construct an emulator-based vaccine that effectively implements array bounds checking at the machine-instruction level. This approach has great potential in catching new vulnerabilities that are being indiscriminately attempted at high volume, as may be the case with an “auto-root” kit or a fast-spreading worm. Since the honeypot is not in the production server’s critical path, its performance is not a primary concern (assuming that attacks are relatively rare phenomena). In the extreme case, we can construct a honeypot using our instruction-level emulator to execute the whole application, although we do not further explore this possibility in this paper.

2.3 Selective Transactional EMulation (STEM)

The recovery mechanism uses an instruction-level emulator, *STEM*, that can be selectively invoked for arbitrary segments of code. This tool permits the execution of emulated and non-emulated code inside the same process. The emulator is implemented as a *C* library that defines special tags (a combination of macros and function calls) that mark the beginning and the end of selective emulation. To use the emulator, we can either link it with an application in advance, or compile it in the code in response to a detected failure, as was done in [29].

Upon entering the vulnerable section of code, the emula-

tor snapshots the program state and executes all instructions on the virtual processor. When the program counter references the first instruction outside the bounds of emulation, the virtual processor copies its internal state back to the real CPU, and lets the program continue execution natively. While registers are explicitly updated, memory updates have implicitly been applied throughout the execution of the emulation. The program, unaware of the instructions executed by the emulator, continues executing directly on the CPU.

To implement fault catching, the emulator simply checks the operands of instructions it is emulating, taking into consideration additional information supplied by the sensor that detected the fault. For example, in the case of division by zero, the emulator need only check the value of the appropriate operand to the *div* instruction. For illegal memory dereferencing, the emulator verifies whether the source or destination addresses of any memory access (or the program counter, for instruction fetches) point to a page that is mapped to the process address space using the *mincore()* system call. Buffer overflow detection is handled by padding the memory surrounding the vulnerable buffer, as identified by the sensor, by one byte, similar to the way StackGuard [13] operates. The emulator then simply watches for memory writes to these memory locations. This approach requires source code availability, so as to insert the “canary” variables. Contrary to StackGuard, our approach allows us to stop the overflow before it overwrites the rest of the stack, and thus to recover the execution. For algorithmic-complexity denial of service attacks, such as the one described in [9], we keep track of the amount of time (in terms of number of instructions) we execute in the instrumented code; if this exceeds a pre-defined threshold, we abort the execution. This threshold may be defined manually, or can be determined by profiling the application under real (or realistic) workloads, although we have not fully explored the possibilities.

We currently assume that the emulator is pre-linked with the vulnerable application, or that the source code of that application is available. It is possible to circumvent this limitation by using the CPU’s programmable breakpoint register (in much the same way that a debugger uses it to capture execution at particular points in the program) to invoke the emulator without the running process even being able to detect that it is now running under an emulator.

2.4 Recovery: Forcing Error Returns

Upon detecting a fault, our recovery mechanism undoes all memory changes and forces an error return from the currently executing function. To determine the appropri-

ate error return value, we analyze the declared type of the function.

Depending on the return type of the emulated function, the system returns an “appropriate” value. This value is determined based on some straightforward heuristics and is placed in the stack frame of the returning function. The emulator then transfers control back to the calling function. For example, if the return type is an *int*, a value of -1 is returned; if the value is *unsigned int* the system returns 0 , *etc.* A special case is used when the function returns a pointer. Instead of blindly returning a *NULL*, we examine if the returned pointer is further dereferenced by the parent function. If so, we expand the scope of the emulation to include the parent function. We handle value-return function arguments similarly. There are some contexts where this heuristic may not work well; however, as a first approach these heuristics worked extremely well in our experiments (see Section 4).

In the future, we plan to use more aggressive source code analysis techniques to determine the return values that are appropriate for a function. Since in many cases a common error-code convention is used in large applications or modules, it may be possible to ask the programmer to provide a short description of this convention as input to our system either through code annotations or as separate input. A similar approach can be used to mark functions that must be fail-safe and should return a specific value when an error return is forced, *e.g.*, code that checks user permissions.

2.5 Caveats and Limitations

While promising, reactive approaches to software faults face a new set of challenges. As this is a relatively unexplored field, some problems are beyond the scope of this paper.

First, our primary goal is to evolve an application protected by STEM towards a state that is highly resistant to exploits and errors. While we expect the downtime for such a system to be reduced, we do not reasonably expect zero downtime. STEM fundamentally relies on the application monitors detecting an error or attack, stopping the application, marking the affected sections for emulated execution, and then restarting the application. This process necessarily involves downtime, but is incurred only once for each detected vulnerability. We believe that combining our approach with microbooting techniques can streamline this process.

A reaction system must evaluate and choose a response from a wide array of choices. Currently, when encountering a fault, a system can (a) crash, (b) crash and be restarted by a monitor [7], (c) return arbitrary values

[26], or (d) slice off the functionality. Most proactive systems take the first approach. We elect to take the last approach. As Section 2.4 shows, this choice seems to work extremely well. This phenomenon also appears at the machine instruction level [33].

However, there is a fundamental problem in choosing a particular response. Since the high-level behavior of any system cannot be algorithmically determined, the system must be careful to avoid cases where the response would take execution down a semantically (from the viewpoint of the programmer’s intent) incorrect path. An example of this type of problem is skipping a check in *sshd* which would allow an otherwise unauthenticated user to gain access to the system. The exploration of ways to bound these types of errors is an open area of research. Our initial approach is to rely on the programmer to provide annotations as to which parts of the code should not be circumvented.

There is a key tradeoff between code coverage (and thus confidence in the level of security the system provides) and performance (processing and memory overhead). Our emulator implementation is a proof of concept; many enhancements are possible to increase performance in a production system. Our main goal is to emphasize the service that such an emulator will provide: the ability to selectively incur the cost of emulation for vulnerable program code only. Our system is directed to these vulnerable sections by runtime sensors – the quality of the application monitors dictates the quality of the code coverage.

Since our emulator is designed to operate at the user level, it hands control to the operating system during system calls. If a fault were to occur in the operating system, our system would not be able to react to it. In a related problem, I/O beyond the machine presents a problem for a rollback strategy. This problem can partially be addressed by the approach taken in [17], by having the application monitors log outgoing data and implementing a callback mechanism for the receiving process.

Finally, in our current work, we assume that the source code of the vulnerable application is available to our system. We briefly discussed how to partially circumvent this limitation in Section 2.3. Additional work is needed to enable our system to work in a binary-only environment.

3 Implementation

We implemented the *STEM x86* emulator to validate the practicality of providing a supervision framework for the feedback control loop through selective emulation of

code slices. Integrating *STEM* into an existing application is straightforward. As shown in Figure 2, four special tags are wrapped around the segment of code that will be emulated.

```
void foo() {
    int a = 1;
    emulate_init();
    emulate_begin(p_args);
    a++;
    emulate_end();
    emulate_term();
    printf("a = %d\n", a);
}
```

Figure 2: A trivial example of using *STEM*. The `emulate_*` calls invoke and terminate execution of *STEM*. The code inside that region is executed by the emulator. In order to illustrate the level of granularity that we can achieve, we show only the increment statement as being executed by the emulator.

The *C* macro `emulate_init()` moves the program state (general, segment, eflags, and FPU registers) into an emulator-accessible global data structure to capture state immediately before *STEM* takes control. The data structure is used to initialize the virtual registers. With the preliminary setup completed, `emulate_begin()` only needs to obtain the memory location of the first instruction following the call to itself. The instruction address is the same as the return address and can be found in the activation record of `emulate_begin()`, four bytes above its base stack pointer.

The fetch/decode/execute/retire cycle of instructions continues until either `emulate_end()` is reached, or when the emulator detects that control is returning to the parent function. If the emulator does not encounter an error during its execution, the emulator’s instruction pointer references the `emulate_term()` macro at completion. To enable the program to continue execution at this address, the return address of the `emulate_begin` activation record is replaced with the current value of the instruction pointer. By executing `emulate_term()`, the emulator’s environment is copied to the program registers and execution continues under normal conditions.

If an exception occurs during emulation, *STEM* locates `emulate_end()` and terminates. Because the emulator saved the state of the program before starting, it can effectively return the program state to its original setting, thus nullifying the effect of the instructions processed through emulation. Essentially, the emulated code is sliced off. At this point, the execution of the code (and its side effects in terms of changes to memory) has been

rolled back.

The emulator is designed to execute in user-mode, so system calls cannot be computed directly without kernel-level permissions. Therefore, when the emulator decodes an interruption with an immediate value of `0x80`, it must release control to the kernel. However, before the kernel can successfully execute the system call, the program state needs to reflect the virtual registers arrived at by *STEM*. Thus, the emulator backs up the real registers and replaces them with its own values. An `INT 0x80` is issued by *STEM*, and the kernel processes the system call. Once control returns to the user-level code, the emulator updates its registers and restores the original values in the program’s registers.

4 Evaluation

Our description of the system raises several questions that need to be answered in order to determine the tradeoffs between effectiveness, practicality, and performance.

1. Can the system detect *real* attacks and faults and react to them ?
2. How effective is our “error virtualization” hypothesis as a recovery mechanism ? Does it work for *real* software ?
3. What is the performance impact of emulation, and what is the gain to be had by using selective emulation ?

In the rest of this section, we provide some preliminary experimental evidence that our system offers a reasonable and *adjustable* tradeoff between the three parameters mentioned above. Naturally, it is impossible to completely cover the space of reactive mechanisms (even within the more limited context of our specific work). Future work is needed to analyze the semantics of error virtualization and the impact that *STEM* has on the security properties of *STEM*-enabled applications. As noted below, we plan to construct a correctness testing framework. However, we believe that our results show that such an approach can work and that additional work is needed to fully explore its capabilities and limitations.

4.1 Effectiveness of Forced Return Recovery

To validate our error virtualization hypothesis using forced function return, introduced in Section 2.4, we experimentally evaluate its effects on program execution on

the Apache *httpd*, OpenSSH *sshd*, and Bind. We run profiled versions of the selected applications against a set of test suites and examine the subsequent call-graphs generated by these tests with *gprof* and Valgrind [21].

The ensuing call trees are analyzed in order to extract leaf functions. The leaf functions are, in turn, employed as potentially vulnerable functions. Armed with the information provided by the call-graphs, we run a script that inserts an early return in all the leaf functions (as described in Section 2.4), simulating an aborted function. Note that these tests do not require going back up the call stack.

In Apache's case, we examined 154 leaf functions. For each aborted function, we monitor the program execution of Apache by running *httperf* [20], a web server performance measurement tool. Success for each test was defined as the application not crashing.

The results from these tests were very encouraging, as 139 of the 154 functions completed the *httperf* tests successfully. In these cases, program execution was not interrupted. What we found to be surprising was that not only did the program not crash, but in some cases *all* the pages were served (as reported by *httperf*). This result is probably because a large number of the functions are used for statistical and logging purposes. Furthermore, out of the 15 functions that produced segmentation faults, 4 did so at start up (and would thus not be relevant in the case of a long-running process). While this result is encouraging, testing the *correctness* of this process would require a regression test suite against the page contents, headers, and HTTP status code for the response. We plan to build this "correctness" testing framework.

Similarly for *sshd*, we iterate through each aborted function while examining program execution during an scp transfer. In the case of *sshd*, we examined 81 leaf functions. Again, the results were auspicious: 72 of the 81 functions maintained program execution. Furthermore, only 4 functions caused segmentation faults; the rest simply did not allow the program to start.

For Bind, we examined the program execution of *named* during the execution of a set of queries; 67 leaf functions were tested. In this case, 59 of the 67 functions maintained the proper execution state. Similar to *sshd*, only 4 functions caused segmentation faults.

These results, along with supporting evidence from [26] and [33], validate our "error virtualization" hypothesis and approach. However, additional work is needed to determine the degree to which other types of applications (*e.g.*, GUI-driven) exhibit the same behavior.

4.2 Attack Exploits

Given the success of our experimental evaluation on program execution, we wanted to further validate our hypothesis against a set of real exploits for Apache, OpenSSH *sshd*, and Bind. No prior knowledge was encoded in our system with respect to the vulnerabilities: for all purposes, this experiment was a zero-day attack.

For Apache, we used the *apache-scalp* exploit that takes advantage of a buffer overflow vulnerability based on the incorrect calculation of the required buffer sizes for chunked encoding requests. We applied selective emulation on the offending function and successfully recovered from the attack; the server successfully served subsequent requests.

The attack used for OpenSSH was the RSAREF2 exploit for SSH-1.2.27. This exploit relies on unchecked offsets that result in a buffer overflow vulnerability. Again, we were able to gracefully recover from the attack and the *sshd* server continued normal operation.

Bind is susceptible to a number of known exploits; for the purposes of this experiment, we tested our approach against the TSIG bug on ISC Bind 8.2.2-x. In the same motif as the previous attacks, this exploit takes advantage of a buffer overflow vulnerability. As before, we were able to safely recover program execution while maintaining service availability.

4.3 Performance

We next turned our attention to the performance impact of our system. In particular, we measured the overhead imposed by the emulator component. *STEM* is meant to be a lightweight mechanism for executing selected portions of an application's code. We can select these code slices according to a number of strategies, as we discussed in Section 2.2.

We evaluated the performance impact of *STEM* by instrumenting the Apache 2.0.49 web server and OpenSSH *sshd*, as well as performing micro-benchmarks on various shell utilities such as *ls*, *cat*, and *cp*.

4.3.1 Testing Environment

The machine we chose to host Apache was a single Pentium III at 1GHz with 512MB of memory running Red-Hat Linux with kernel 2.4.20. The machine was under a light load during testing (standard set of background applications and an X11 server). The client machine was a dual Pentium II at 350 MHz with 256MB of memory running RedHat Linux 8.0 with kernel 2.4.18smp. The client

machine was running a light load (X11 server, *sshd*, background applications) in addition to the test tool. Both emulated and non-emulated versions of Apache were compiled with the `-enable-static-support` configuration option. Finally, the standard runtime configuration for Apache 2.0.49 was used; the only change we made was to enable the *server-status* module (which is compiled in by default but not enabled in the default configuration). *STEM* was compiled with the “`-g -static -fno-defer-pop`” flags. In order to simplify our debugging efforts, we did not include optimization.

We chose the Apache *flood* *httpd* testing tool to evaluate how quickly both the non-emulated and emulated versions of Apache would respond and process requests. In our experiments, we chose to measure performance by the total number of requests processed, as reflected in Figures 3 and 4. The value for total number of requests per second is extrapolated (by *flood*’s reporting tool) from a smaller number of requests sent and processed within a smaller time slice; the value should not be interpreted to mean that our test Apache instances and our test hardware actually served some 6000 requests per second.

4.3.2 Emulation of Apache Inside Valgrind

To get a sense of the performance degradation imposed by running the entire system inside an emulator other than *STEM*, we tested Apache running in Valgrind version 2.0.0 on the Linux test machine that hosted Apache for our *STEM* test trials.

Valgrind has two notable features that improve performance over our full emulation of the main request loop. First, Valgrind maintains a 14 MB cache of translated instructions which are executed natively after the first time they are emulated, while *STEM* always translates each encountered instruction. Second, Valgrind performs some internal optimizations to avoid redundant load, store, and register-to-register move operations.

We ran Apache under Valgrind with the default skin *Memcheck* and tracing all children processes. While Valgrind performed better than our emulation of the full request processing loop, it did not perform as well as our emulated slices, as shown in Figure 3 and the timing performance in Table 1.

Finally, the Valgrind-ized version of Apache is 10 times the size of the regular Apache image, while Apache with *STEM* is not noticeably larger.

4.3.3 Full Emulation and Baseline Performance

We demonstrate that emulating the bulk of an application entails a significant performance impact. In particular, we emulated the main request processing loop for Apache (contained in *ap_process_http_connection()*) and compared our results against a non-emulated Apache instance. In this experiment, the emulator executed roughly 213,000 instructions. The impact on performance is clearly seen in Figure 3 and further elucidated in Figure 4, which plots the performance of the fully emulated request-handling procedure.

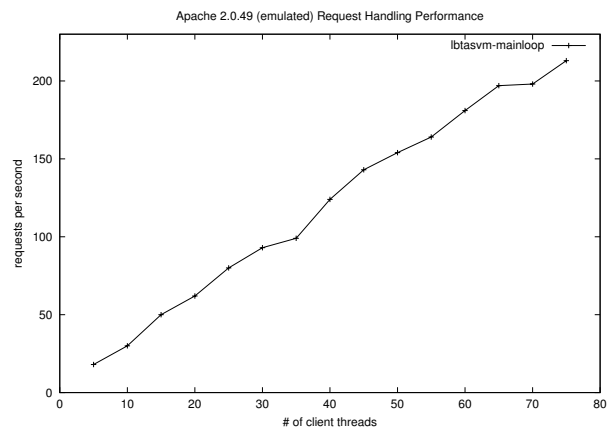


Figure 4: A closer look at the performance for the fully emulated version of main processing loop. While there is a considerable performance impact compared to the non-emulated request handling loop, the emulator appears to scale at the characteristic linear rate, indicating that it does not create additional overhead beyond the cost of emulation.

In order to get a more complete sense of this performance impact, we timed the execution of the request handling procedure for both the non-emulated and fully-emulated versions of Apache by embedding calls to *gettimeofday()* where the emulation functions were (or would be) invoked.

For our test machines and sample loads, Apache normally (*e.g.*, non-emulated) spent 6.3 milliseconds to perform the work in the *ap_process_http_connection()* function, as shown in Table 1. The fully instrumented loop running in the emulator spends an average of 278 milliseconds per request in that particular code section. For comparison, we also timed Valgrind’s execution of this section of code; after a large initial cost (to perform the initial translation and fill the internal instruction cache) Valgrind executes the section with a 34 millisecond average. These initial costs sometimes exceeded one or two seconds; we ignore them in our data and measure Val-

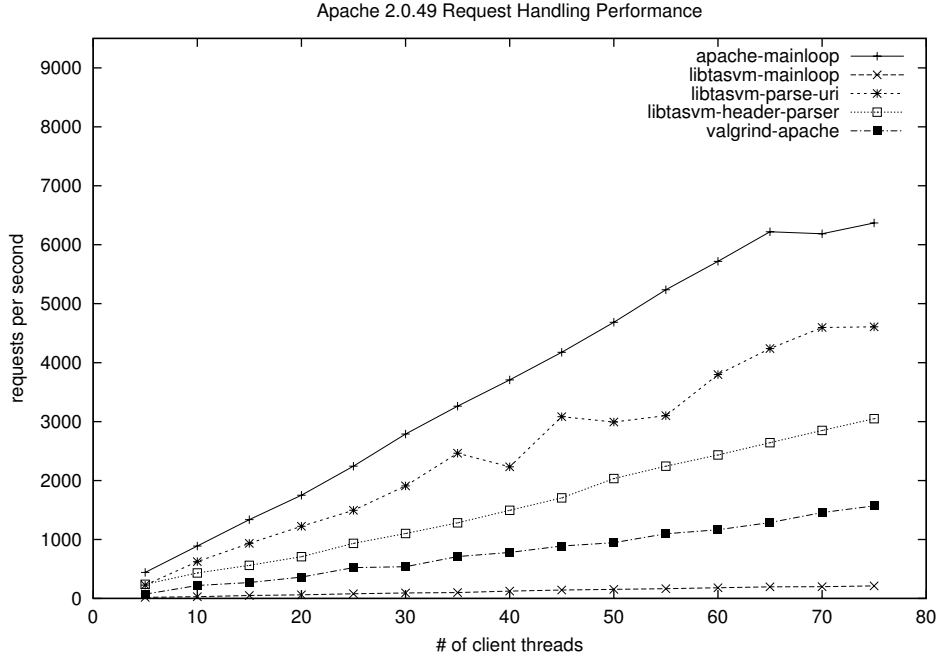


Figure 3: Performance of the system under various levels of emulation. This data set includes Valgrind for reference. While full emulation is fairly expensive, selective emulation of input handling routines appears quite sustainable. Valgrind runs better than *STEM* when executing the entire request loop. As expected, selective emulation still performs better than Valgrind.

grind only after it has stabilized.

Apache	trials	Mean	Std. Dev.
Normal	18	6314	847
STEM	18	277927	74488
Valgrind	18	34192	11204

Table 1: Timing of main request processing loop. Times are in microseconds. This table shows the overhead of running the whole primary request handling mechanism inside the emulator. In each trial a user thread issued an HTTP GET request.

4.3.4 Selective Emulation

In order to identify possible vulnerable sections of code in Apache 2.0.49, we used the RATS tool. The tool identified roughly 270 candidate lines of code, the majority of which contained fixed size local buffers. We then correlated the entries on the list with code that was in the primary execution path of the request processing loop. The two functions that are measured perform work on input that is under client control, and are thus likely candidates for attack vectors.

The main request handling logic in Apache 2.0.49 begins in the `ap_process_http_connection()` function. The effective work of this function is carried out by two subroutines: `ap_read_request()` and `ap_process_request()`. The `ap_process_request()` function is where Apache spends most of its time during the handling of a particular request. In contrast, the `ap_read_request()` function accounts for a smaller fraction of the request handling work. We chose to emulate subroutines of each function in order to assess the impact of selective emulation.

We constructed a partial call tree and chose the `ap_parse_uri()` function (invoked via `read_request_line()` in `ap_read_request()`) and the `ap_run_header_parser()` function (invoked via `ap_process_request_internal()` in `ap_process_request()`). The emulator processed approximately 358 and 3229 instructions, respectively, for these two functions. In each case, the performance impact, as expected, was much less than the overhead incurred by needlessly emulating the entire work of the request processing loop.

4.3.5 Microbenchmarks

Using the client machine from the Apache performance tests, we ran a number of micro-benchmarks to gain a

broader view of the performance impact of *STEM*. We selected some common shell utilities and measured their performance for large workloads running both with and without *STEM*.

For example, we issued an `'ls -R'` command on the root of the Apache source code with both *stderr* and *stdout* redirected to */dev/null* in order to reduce the effects of screen I/O. We then used *cat* and *cp* on a large file (also with any screen output redirected to */dev/null*). Table 2 shows the result of these measurements.

As expected, there is a large impact on performance when emulating the majority of an application. Our experiments demonstrate that only emulating potentially vulnerable sections of code offers a significant advantage over emulation of the entire system.

5 Related Work

Modeling executing software as a transaction that can be aborted has been examined in the context of language-based runtime systems (namely, Java) in [28, 27]. That work, focused on safely terminating misbehaving threads, introduces the concept of “soft termination.” Soft termination allows thread termination while preserving the stability of the language runtime, without imposing unreasonable performance overheads. In that approach, threads (or *codelets*) are each executed in self-encompassing transactions, applying standard ACID semantics. This allows changes to the runtime’s (and other threads’) state made by the terminated codelet to be rolled back. The performance overhead of that system can range from 200% up to 2300%. Relative to that work, our contribution is twofold. First, we apply the transactional model to an unsafe language such as *C*, addressing several (but not all) challenges presented by that environment. Second, by selectively emulating, we substantially reduce the performance overhead of the application. However, there is no free lunch: this reduction comes at the cost of allowing failures to occur. Our system aims to automatically evolve a piece of code such that it *eventually* (i.e., once an attack has been observed, possibly more than once) does not succumb to attacks.

Oplinger and Lam propose [23] another transactional approach to improve software reliability. Their key idea is to employ thread level speculation (TLS) and execute an application’s monitoring code in parallel with the primary computation. The computation “transaction” is rolled back depending on the results of the monitoring code.

Virtual machine emulation of operating systems or processor architectures to provide a sandboxed environment

is an active area of research. Virtual machine monitors (VMM) are employed in a number of security-related contexts, from autonomic patching of vulnerabilities [29] to intrusion detection [14].

Other protection mechanisms include compiler techniques like StackGuard [13] and safer libraries, such as *libsafe* and *libverify* [4]. Other tools exist to verify and supervise code during development or debugging. Of these tools, Purify and Valgrind [21] are popular choices.

Valgrind is a program supervision framework that enables in-depth instrumentation and analysis of IA-32 binaries without recompilation. Valgrind has been used by Barrantes *et al.* [5] to implement instruction set randomization techniques to protect programs against code insertion attacks. Other work on instruction-set randomization includes [16], which employs the i386 emulator Bochs.

Program shepherding [19] is a technique developed by Kiriansky, Bruening, and Amarasinghe. The authors describe a system based on the RIO [11] architecture for protecting and validating control flows according to some security policy without modification of IA-32 binaries for Linux and Windows. The system works by validating branch instructions and storing the decision in a cache, thus incurring little overhead.

The work by Dunlap, King, Cinar, Basrai, and Chen [12] is closely related to the work presented in this paper. ReVirt is a system implemented in a VMM that logs detailed execution information. This detailed execution trace includes non-deterministic events such as timer interrupt information and user input. Because ReVirt is implemented in a VMM, it is more resistant to attack or subversion. However, ReVirt’s primary use is as a forensic tool to replay the events of an attack, while the goal of *STEM* is to provide a lightweight and minimally intrusive mechanism for protecting code against malicious input *at runtime*.

King, Dunlap, and Chen [18] discuss optimizations that reduce the performance penalties involved in using VMMs. There are three basic optimizations: reduce the number of context switches by moving the VMM into the kernel, reduce the number of page faults by allowing each VMM process greater freedom in allocating and maintaining address space, and ameliorate the penalty for switching between guest kernel mode and guest user mode by simply changing the bounds on the guest memory area rather than re-mapping.

An interesting application of ReVirt [12] is BackTracker [17], a tool that can automatically identify the steps involved in an intrusion. Because detailed execution information is logged, a dependency graph can be constructed backward from the detection point to provide forensic in-

Test Type	trials	mean (s)	Std. Dev.	Min	Max	Instr. Emulated
ls (non-emu)	25	0.12	0.009	0.121	0.167	0
ls (emu)	25	42.32	0.182	42.19	43.012	18,000,000
cp (non-emu)	25	16.63	0.707	15.80	17.61	0
cp (emu)	25	21.45	0.871	20.31	23.42	2,100,000
cat (non-emu)	25	7.56	0.05	7.48	7.65	0
cat (emu)	25	8.75	0.08	8.64	8.99	947,892

Table 2: Microbenchmark performance times for various command line utilities.

formation about an attack.

Toth and Kruegel [32] propose to detect buffer overflow payloads (including previously unseen ones) by treating inputs received over the network as code fragments. They show that legitimate requests will appear to contain relatively short sequences of valid x86 instruction opcodes, compared to attacks that will contain long sequences. They integrate this mechanism into the Apache web server, resulting in a small performance degradation.

Some interesting work has been done to deal with memory errors at runtime. For example, Rinard *et al.* [25] have developed a compiler that inserts code to deal with writes to unallocated memory by automatically expanding the target buffer. Such a capability aims toward the same goal our system does: provide a more robust fault response rather than simply crashing. The technique presented in [25] is modified in [26] and introduced as *failure-oblivious computing*. This behavior of this technique is close to the behavior of our system.

One of the most critical concerns with recovering from software faults and vulnerability exploits is ensuring the consistency and correctness of program data and state. An important contribution in this area is presented by Dempsy [10], which discusses mechanisms for detecting corrupted data structures and fixing them to match some pre-specified constraints. While the precision of the fixes with respect to the semantics of the program is not guaranteed, their test cases continued to operate when faults were randomly injected.

Suh *et al.* [31] propose a hardware based solution that can be used to thwart control-transfer attacks and restrict executable instructions by monitoring “tainted” input data. In order to identify “tainted” data, they rely on the operating system. If the processor detects the use of this tainted data as a jump address or an executed instruction, it raises an exception that can be handled by the operating system. The authors do not address the issue of recovering program execution and suggest the immediate termination of the offending process. DIRA [30] is a technique for automatic detection, identification and repair of control-hijacking attacks. This solution is imple-

mented as a GCC compiler extension that transforms a program’s source code and adds heavy instrumentation so that the resulting program can perform these tasks. The use of checkpoints throughout the program ensures that corruption of state can be detected if control sensitive data structures are overwritten. Unfortunately, the performance implications of the system make it unusable as a front line defense mechanism. Song and Newsome [22] propose dynamic taint analysis for automatic detection of overwrite attacks. Tainted data is monitored throughout the program execution and modified buffers with tainted information will result in protection faults. Once an attack has been identified, signatures are generated using automatic semantic analysis. The technique is implemented as an extension to Valgrind and does not require any modifications to the program’s source code but suffers from severe performance degradation.

While our prototype x86 emulator is a fairly straightforward implementation, it can gain further performance benefits by using Valgrind’s technique of caching already translated instructions. With some further optimizations, *STEM* is a viable and practical approach to protecting code. In fact, [6] outlines several ways to optimize emulators; their approaches reduce the performance overhead (as measured by two SPEC2000 benchmarks, *crafty* and *vpr*) from a factor of 300 to about 1.7. Their optimizations include caching basic blocks (essentially what VG is doing), linking direct and indirect branches, and building traces.

6 Conclusions

Software errors and the concomitant potential for exploitable vulnerabilities remain a pervasive problem. Accepted approaches to this problem are almost always proactive, but it seems unlikely that such strategies will result in error-free code. In the absence of such guarantees, reactive techniques for error toleration and recovery can be powerful tools.

We have described a lightweight mechanism for super-

vising the execution of an application that has already exhibited a fault and preventing its recurrence. Our work aims to ultimately create a “self-healing” system. We use selective emulation of the code immediately surrounding a detected fault to validate the operands to machine instructions, as appropriate for the type of fault; we currently handle buffer overflows, illegal memory dereferences, divide-by-zero exceptions, and some types of algorithmic-complexity denial of service attacks. Once a fault has been detected, we restore control to a safe flow by forcing the function containing the fault to return an error value and rolling back any memory modifications the emulated code has made during its execution.

Our intuition is that most applications are written well enough to catch the majority of errors, but fail to consider some boundary conditions that allow the fault to manifest itself. By catching these extreme cases and returning an error, we make use of the already existing error-handling code. We validate this hypothesis using a set of real attacks, as well as randomly induced faults in some widely used open-source servers (Apache, *sshd*, and *Bind*). Our results show that our system works in over 88% of all cases, allowing the application to continue execution and behave correctly. Furthermore, by using selective emulation of small code segments, we minimize the performance impact on production servers.

Our approach is a first exploration into a reactive system that allows quick, *automated* reaction to software failures, thereby increasing service availability in the presence of general software bugs. We re-emphasize that our approach can be used to catch a variety of software failures, not just malicious attacks. Our plans for future work include enhancing the performance of our prototype emulator and further validating our “error virtualization” hypothesis by extending the number of applications and attacks examined.

References

- [1] CERT Advisory CA-2003-21: W32/Blaster Worm. <http://www.cert.org/advisories/CA-2003-20.html>, August 2003.
- [2] The Spread of the Sapphire/Slammer Worm. <http://www.silicondefense.com/research/worms/slammer.php>, February 2003.
- [3] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.
- [4] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.
- [5] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Distrust Binary Code Injection Attacks. In *10th ACM Conference on Computer and Communications Security (CCS)*, October 2003.
- [6] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275, 2003.
- [7] G. Candea and A. Fox. Crash-Only Software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.
- [8] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 235–244, November 2002.
- [9] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, August 2003.
- [10] B. Demsky and M. C. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, October 2003.
- [11] E. Duesterwald and S. P. Amarasinghe. On the Run – Building Dynamic Program Modifiers for Optimization, Introspection, and Security. In *Conference on Programming Language Design and Implementation (PLDI)*, 2002.
- [12] G. W. Dunlap, S. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, February 2002.
- [13] C. C. et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

- [14] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *10th ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2003.
- [15] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, June 2002.
- [16] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *10th ACM Conference on Computer and Communications Security (CCS)*, October 2003.
- [17] S. T. King and P. M. Chen. Backtracking Intrusions. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.
- [18] S. T. King, G. Dunlap, and P. Chen. Operating System Support for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.
- [19] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.
- [20] D. Mosberger and T. Jin. httpperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59–67. ACM, June 1998.
- [21] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, 2003.
- [22] J. Newsome and D. Dong. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *The 12th Annual Network and Distributed System Security Symposium*, February 2005.
- [23] J. Oplinger and M. S. Lam. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002.
- [24] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, August 2003.
- [25] M. Rinard, C. Cadar, D. Dumitran, D. Roy, and T. Leu. A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors). In *Proceedings 20th Annual Computer Security Applications Conference (ACSAC)*, December 2004.
- [26] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W. Beebe. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.
- [27] A. Rudys and D. S. Wallach. Transactional Roll-back for Language-Based Systems. In *ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2001.
- [28] A. Rudys and D. S. Wallach. Termination in Language-based Systems. *ACM Transactions on Information and System Security*, 5(2), May 2002.
- [29] S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE Workshop on Enterprise Technologies: Infrastructure for Collaborative Enterprises (WET-ICE), Workshop on Enterprise Security*, pages 220–225, June 2003.
- [30] A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *The 12th Annual Network and Distributed System Security Symposium*, February 2005.
- [31] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *SIGOPS Oper. Syst. Rev.*, 38(5):85–96, 2004.
- [32] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID)*, October 2002.
- [33] N. Wang, M. Fertig, and S. Patel. Y-Branched: When You Come to a Fork in the Road, Take It. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, September 2003.
- [34] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of ACM SOSP*, October 2003.

Attrition Defenses for a Peer-to-Peer Digital Preservation System

TJ Giuli Petros Maniatis Mary Baker
Stanford University, CA Intel Research, Berkeley, CA HP Labs, Palo Alto, CA

David S. H. Rosenthal Mema Roussopoulos
Stanford University Libraries, CA Harvard University, Cambridge, MA

Abstract

In peer-to-peer systems, *attrition attacks* include both traditional, network-level denial of service attacks as well as application-level attacks in which *malign* peers conspire to waste *loyal* peers' resources. We describe several defenses for the LOCKSS peer-to-peer digital preservation system that help ensure that application-level attrition attacks even from powerful adversaries are less effective than simple network-level attacks, and that network-level attacks must be intense, widespread, and prolonged to impair the system.

1 Introduction

Denial of Service (DoS) attacks are among the most difficult for distributed systems to resist. Distinguishing legitimate requests for service from the attacker's requests can be tricky, and devoting substantial effort to doing so can easily be self-defeating. The term DoS was introduced by Needham [34] with a broad meaning but over time it has come to mean high-bit-rate network-level flooding attacks [23] that rapidly degrade the usefulness of the victim system. In addition to DoS, we use the term *attrition* to include also moderate- or low-bit-rate application-level attacks that impair the victim system.

The mechanisms described in this paper are aimed at equipping the LOCKSS¹ (Lots Of Copies Keep Stuff Safe) peer-to-peer (P2P) digital preservation system to resist attrition attacks. The system is in use at about 80 libraries worldwide; publishers of about 2000 titles have endorsed its use. Cooperation among peers reduces the cost and increases the reliability of preservation, eliminates the need for backup, and greatly reduces other operator interventions.

A *loyal* (non-malign) peer participates in the LOCKSS system for two reasons: to achieve regular reassurance that its content agrees with the consensus of the peers holding copies of the same content, and if it does not,

to obtain the needed repair. The goal of an attrition adversary is to prevent loyal peers from successfully determining the consensus of their peers or from obtaining requested repairs for so long that undetected storage problems such as natural "bit rot" or human error corrupt their content. Other types of resource waste may be inconvenient but have no lasting effect on this system.

In prior work [30] we defended LOCKSS peers against attacks seeking to corrupt their content. That system, however, remained vulnerable to application-level attrition; about 50 malign peers could abuse the protocol to prevent a network of 1000 peers from auditing and repairing their content.

We have developed a set of defenses, some adapted from other systems, whose combination in a P2P context provides novel and effective protection against attrition. These defenses include *admission control*, *desynchronization*, and *redundancy*. Admission control, effected via rate limitation, first-hand reputation, and effort balancing, ensures that legitimate requests can be serviced even during malicious request floods. Desynchronization ensures that progress continues even if some suppliers of a needed service are currently too busy. Redundancy ensures that the attacker cannot incapacitate the system by targeting only few peers at a time. Our defenses may not all be immediately applicable to all P2P applications, but we believe that many systems may benefit from a subset of these defenses, and that our analysis of the effectiveness of these defenses is more broadly useful.

This paper presents a new design of the LOCKSS protocol that makes four contributions. First, we demonstrate via simulation how our new design ensures that application-level attrition, no matter how powerful the attacker, is less effective than simple network flooding. We do this while retaining our previous resistance against other adversaries. Second, we show that even network-level flooding attacks that continuously prevent *all* communication among a majority of the peers must last for months to affect the system significantly. Such attacks

are orders of magnitude more powerful than those observed in practice [33]. Third, since resource management lies at the crux of attrition attacks and their defenses, we extend our prior evaluation [30] to deal with numerous concurrently preserved archival units of content competing with each other for resources. Finally, resource over-provisioning is essential in defending against attrition attacks. We show that with a practical amount of over-provisioning we can defend the LOCKSS system from an arbitrarily powerful attrition adversary.

In the rest of this paper, we first describe our application. We continue by outlining how we would like this application to behave under different levels of attrition attack. We give an overview of the LOCKSS protocol, describing how it incorporates each of our attrition defenses. We then explain the results of a systematic exploration of simulated attacks against the resulting design, showing that it successfully defends against attrition attacks at all layers, from the network level up through the application protocol. Finally, we describe how the new LOCKSS protocol compares to our previous work, as well as other related work.

2 The Application

In this section, we provide an overview of the digital preservation problem for academic publishing. We then present and justify the set of design goals required of any solution to this problem, setting the stage for the LOCKSS approach in subsequent sections.

Academic publishing has migrated to the Web [46], placing society's scientific and cultural heritage at a variety of risks such as confused provenance, accidental editing by the publisher, storage corruption, failed backups, government or corporate censorship, and vandalism. The LOCKSS system was designed [39] to provide librarians with the tools they need to preserve their community's access to journals and other Web materials.

Any solution must meet six stringent requirements. First, since under U.S. law copyrighted Web content can only be preserved with the owner's permission [16], the solution must accommodate the publishers' interests. Requiring publishers, for example, to offer perpetual no-fee access or digital signatures on content makes them reluctant to give that permission. Second, a solution must be extremely cheap in terms of hardware, operating cost, and human expertise. Few libraries could afford [3] a solution involving handling and securely storing off-line media, but most can afford the few cheap off-the-shelf PCs that provide sufficient storage for tens of thousands of journal-years. Third, the existence of cheap, reliable storage cannot be assumed; affordable storage is unreliable [22, 38]. Fourth, a solution must have a long time horizon. Auditing content against stored dig-

ital signatures, for example, assumes not only that the cryptosystem will remain unbroken, but also that the secrecy, integrity, and availability of the keys are guaranteed for decades. Fifth, a solution must anticipate adversaries capable of powerful attacks sustained over long periods; it must withstand these attacks, or at least degrade slowly and gracefully while providing unambiguous warnings [37]. Sixth, a solution must not require a central locus of control or administration, if it is to withstand concentrated technical or legal attacks.

Two different architectures have been proposed for preserving Web journals. The centralized architecture of a "trusted third party" archive requires publishers to grant a third party permission, under certain circumstances, to republish their content. Obtaining this permission involves formidable legal and business obstacles [5]. In contrast, the distributed architecture of the LOCKSS system consists of many individual archives at subscribing (second party) libraries. Readers only access their local library's copy, whose subscription already provides them access to the publisher's copy. Most publishers see this as less of a risk to their business, and are willing to add this permission to the subscription agreement. It is thus important to note that our goal is not to minimize the number of replicas consistent with content safety. Instead, we strive to minimize the per-replica cost of maintaining a large number of replicas. We trade extra replicas for fewer lawyers, an easy decision given their relative costs.

The LOCKSS design is extremely conservative, making few assumptions about the infrastructure. Although we believe this is appropriate for a digital preservation system, less conservative assumptions are certainly possible. Increasing risk can increase the amount of content that can be preserved with given computational power. Limited amounts of reliable, write-once memory would allow audits against local hashes, a reliable public key infrastructure might allow publishers to sign their content and peers to audit against the signatures, and so on. Conservatively, the assumptions underlying such optimizations could be violated without warning at any time; the write-once memory might be corrupted or mishandled, or a private key might leak. Thus, these optimizations still require a distributed audit mechanism as a fallback. The more a peer operator can do to avoid local failures the better the system works, but our conservative design principles lead us to focus on mechanisms that minimize dependence on these efforts.

With the application of digital preservation for academic publishing in mind, we tackle the "abstract" problem of auditing and repairing replicas of distinct *archival units* or AUs (a year's run of an on-line journal, in our target application) preserved by a population of peers (libraries) in the face of attrition attacks. For each AU it

preserves, a peer starts out with its own, correct replica (obtained from the publisher's Web site), which it can only use to satisfy local read requests (from local patrons) and to assist other peers with replica repairs. In the rest of this paper we refer to AUs, peers, and replicas, rather than journals and libraries.

3 System Model

In this section we present the adversary we model, our security goals and the framework for our defenses.

3.1 Adversary Model

Our conservative design philosophy leads us to assume a powerful adversary with several important abilities. *Pipe stoppage* is his ability to prevent communication with victim peers for extended periods by flooding links with garbage packets or using more sophisticated techniques [26]. *Total information awareness* allows him to control and monitor all of his resources instantaneously. He has *unconstrained identities* in that he can purchase or spoof unlimited network identities. *Insider information* provides him complete knowledge of victims' system parameters and resource commitments. *Masquerading* means that loyal peers cannot detect him, as long as he follows the protocol. Finally, he has *unlimited computational resources*, though he is polynomially bounded in his computations (i.e., he cannot invert cryptographic functions).

The adversary employs these capabilities in *effortless* and *effortful* attacks. An effortless attack requires no measurable computational effort from the attacker and includes traditional DoS attacks such as pipe stoppage. An effortful attack requires the attacker to invest in the system with computational effort.

3.2 Security Goals

The overall goals of the LOCKSS system are that, with high probability, the consensus of peers reflects the correct AU, and readers access good data. In contrast, an attrition adversary's goal is to decrease significantly the probability of these events by preventing peers from auditing their replicas for a long time, long enough for undetected storage problems such as "bit rot" to occur.

Severe but narrowly focused pipe stoppage attacks in the wild last for days or weeks [33]. Our goal is to ensure that, in the very least, the LOCKSS system withstands or degrades gracefully with even broader such attacks sustained over months. Beyond pipe stoppage, attackers must use protocol messages to some extent. We seek to ensure the following three conditions. First, a peer manages its resources so as to prevent exhaustion no matter

how much effort is exerted by however many identities requesting service. Second, when deciding which requests to service, a peer gives preference to requests from those likely to behave properly (i.e., "ostensibly legitimate"). And third, at every stage of a protocol exchange, an ostensibly legitimate attacker expends commensurate effort to that which he imposes upon the defenders.

3.3 Defensive Framework

We seek to curb the adversary's success by modeling a peer's processing of inbound messages as a series of filters, each costing a certain amount to apply. A message rejected by a filter has no further effect on the peer, allowing us to estimate the cost of eliminating whole classes of messages from further consideration. Each filter increases the effort a victim needs to defend itself, but limits the effectiveness of some adversary capability. The series of filters as a whole is *sound* if the cost of applying a filter to the input stream passed through its preceding filter is low enough to permit the system to make progress. The filters include a volume filter, a reciprocity filter, and a series of effort filters.

The *volume filter* models a peer's network connection. It represents the physical limits on the rate of inbound messages that an adversary can force upon the peer. It is an unavoidable filter; no adversary can push data through a victim's network card at a rate greater than the card's throughput. Soundness requires the volume filter to restrict the volume of messages enough that processing costs at the next filter downstream are low. This condition can be enforced either through traffic shaping or via the low-tech choice of configuring peers with low-bandwidth network cards.

The *reciprocity filter* takes inbound messages at the maximum rate exported by the volume filter and further limits them by rejecting those sent from peers who appear to be misbehaving. A peer's reciprocity filter favors those of its peers who engage it with requests at the same average rate as it engages them. The filter further penalizes those peers it has not known for long enough to evaluate their behavior. In this sense, the reciprocity filter implements a *self-clocking* invariant, by which inbound traffic exiting the filter mirrors in volume traffic originated at the peer. Thus on average the *number* of requests passed to the next filter matches the number of requests inflicted by the peer upon others.

The *effort filters* focus on the balance of effort expended by the peer and a correspondent peer while the two are cooperating on an individual content audit request. These filters ensure that the computational effort imposed upon a potential victim peer by its ostensibly legitimate correspondent is matched by commensurate effort borne by that correspondent. For example, an at-

tacker can only trick its victim peer into cryptographically hashing large amounts of data by first performing the same hash itself (or other effort equivalent to the same hash). As a result, these filters enforce the invariant that ostensible legitimacy costs the attacker as much as it allows the attacker to inflict on its victim. Furthermore, the effort filters ensure that a peer can detect at a low cost that an attacker has abandoned ostensible legitimacy.

In summary, these filters take an input stream of protocol messages and reduce it to levels consistent with legitimate traffic in terms of volume (volume filter), then in number of individual messages per source (reciprocity filter), and then in effort induced per message (effort filters). Malicious interactions that pass all filters can ultimately affect the victim peer adversely, but are ensured to impose no more than manageable additional burden on the victim peer and are guaranteed to cost the attacker as much burden in the process. The former guarantee is essential for the correct operation of good peers in all cases, whereas the latter is only meaningful when the adversary is resource-constrained.

We show in Section 7.4 that the most effective strategy for effortful attacks is to emulate legitimacy, and that even this has minimal effect on the utility of the system. Effortless attacks, such as traditional distributed DoS (DDoS) attacks, are more effective but must be maintained for a long time against most of the peer population to degrade the system significantly (Section 7.2).

4 The LOCKSS Replica Auditing and Repair Protocol

The LOCKSS audit process is a sequence of “opinion polls” conducted by every peer on each of its AU replicas. At intervals, typically every 3 months, a peer (the *poller*) picks a random sample of peers that it knows to be preserving an AU, and invites those peers as *voters* into a poll. Each voter individually hashes a poller-supplied nonce and its replica of the AU to produce a fresh vote, which the poller tallies. If the poller is outvoted in a landslide (e.g., it disagrees with 80% of the votes), it assumes its replica is corrupt and repairs it from a disagreeing voter. The roles of poller and voter are distinct, but every peer plays both.

The general structure of a poll follows the timeline of Figure 1. A poll consists of two phases: *vote solicitation* and *evaluation*. In the vote solicitation phase the poller requests and obtains votes from as many voters in its sample of the population as possible. Then the poller begins the evaluation phase, during which it compares these votes to its own replica, one hashed content block at a time, and tallies them. If the hashes disagree the poller may request repair blocks from its voters and reevaluate

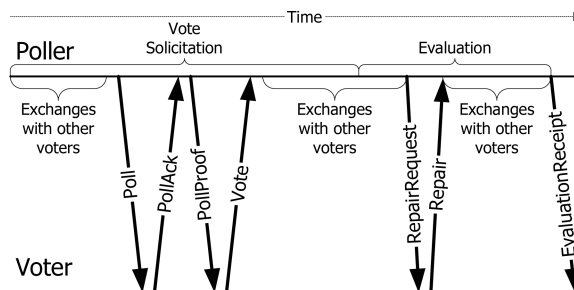


Figure 1: A timeline of a poll, showing the message exchange between the poller and a voter.

the block. If in the eventual tally, after any repairs, the poller agrees with the landslide majority, it sends a receipt to each of its voters and immediately starts a new poll. Peers interleave progress on their own polls with voting in other peers’ polls, spreading each poll over a period chosen so that polls on a given AU occur at a rate much higher than that of undetected storage errors.

4.1 Vote Solicitation

The outcome of a poll is determined by the votes of the *inner circle* peers, chosen at the start of the poll by the poller from its *reference list* for the AU. The reference list contains mostly peers that have agreed with the poller in recent polls on the AU, and a few peers from its static *friends list*, maintained by the poller’s operator.

A poll is considered successful if its result is based on a minimum number of inner circle votes, the *quorum*, which is typically 10, but may change according to the application’s needs for fault tolerance. To ensure that a poll is likely to succeed, a poller invites into its poll a larger inner circle than the quorum (typically, twice as large). If at first try an inner circle peer fails to respond to an invitation, or refuses it, the poller contacts a different inner circle voter, retrying the reluctant peer later in the same vote solicitation phase.

An individual vote solicitation consists of four messages (see Figure 1): Poll, PollAck, PollProof, and Vote. For the duration of a poll, a poller establishes an encrypted TLS session with each voter individually, via an anonymous Diffie-Hellman key exchange. Every protocol message is conveyed over this TLS session, either keeping the same TCP connection from message to message, or resuming the TLS session over a new one.

The Poll message invites a voter to participate in a poll on an AU. The invited peer responds with a PollAck message, indicating either a refusal to participate in the poll at the time, or an acceptance of the invitation, if it can compute a vote within a predetermined time allowance. The voter commits and reserves local resources to that ef-

fect. The PollProof message supplies the voter with a random nonce to be used during vote construction. To compute its vote, the voter uses a cryptographic hash function to hash the nonce supplied by the poller, followed by its replica of the AU, block by block. The vote consists of the running hashes produced at each block boundary. Finally, the voter sends its vote back to the poller in a Vote message.

These messages also contain proofs of computational effort, such as those introduced by Dwork et al. [15], sufficient to ensure that, at every protocol stage, the requester of a service has more invested in the exchange than the supplier of the service (see Section 5.1).

4.2 Peer Discovery

The poller uses the vote solicitation phase of a poll not only to obtain votes for the current poll, but also to discover new peers for its reference list from which it can solicit inner circle votes in future polls.

Discovery is effected via *nominations* included in Vote messages. A voter picks a random subset of its current reference list, which it includes in the Vote message. The poller accumulates these nominations. When it concludes its inner circle solicitations, it chooses a random sample of these nominations as its *outer circle*. It proceeds to solicit regular votes from these outer circle peers in a manner identical to that used for inner circle peers.

The purpose of the votes obtained from outer circle voters is to show the “good behavior” of newly discovered peers. Those who perform correctly, by supplying votes that agree with the prevailing outcome of the poll, are added into the poller’s reference list at the conclusion of the poll; the outcome of the poll is computed only from inner-circle votes.

4.3 Vote Evaluation

Once the poller has accumulated all votes it could obtain from inner and outer circle voters, it begins the poll’s evaluation phase. During this phase, the poller computes, in parallel, all block hashes that each voter *should have* computed, if that voter’s replica agreed with the poller’s. A vote *agrees* with the poller on a block if the hash in the vote and that computed by the poller are the same.

For each hash computed by the poller for an AU block, there are three possibilities: first, the landslide majority of inner-circle votes (e.g., 80%) agree with the poller; in this case, the poller considers the audit successful up to this block and proceeds with the next block. Second, the landslide majority of inner-circle votes disagree with the poller; in this case, the poller regards its own replica of the AU as damaged, obtains a repair from one of the disagreeing voters (via the RepairRequest and Repair mes-

sages), and reevaluates the block hoping to find itself in the landslide majority, as above. Third, if there is no landslide majority of agreeing or disagreeing votes, the poller deems the poll inconclusive, raising an alarm that requires attention from a human operator.

Throughout the evaluation phase, the poller may also decide to obtain a repair from a random voter, even if one is not required (i.e., even if the corresponding block met with a landslide agreement). The purpose of such *frivolous* repairs is to prevent targeted free-riding via the refusal of repairs; voters are expected to supply a small number of repairs once they commit to participate in a poll, and are penalized otherwise (Section 5.1).

If the poller hashes all AU blocks without raising an alarm, it concludes the poll by sending an evaluation receipt to each voter (with an EvaluationReceipt message), containing cryptographic proof that it has evaluated received votes. The poller then updates its reference list by removing all voters whose votes determined the poll outcome and by inserting all agreeing outer-circle voters and some peers from the friends list (for details see [30]). The poller then restarts a poll on the same AU, scheduling it to conclude one interpoll interval into the future.

5 LOCKSS Defenses

Here we outline the attrition defenses of the LOCKSS protocol: admission control, desynchronization, and redundancy. These defenses raise system costs for both loyal peers and attackers, but favor ostensible legitimacy. Given a constant amount of over-provisioning, loyal peers continue to operate at the necessary rate regardless of the attacker’s power. Many systems over-provision resources to protect performance from known worst-case behavior (e.g., the Unix file system [31]).

In prior work [30] we applied some of these defenses (such as redundancy and some aspects of admission control, including rate limitation and effort balancing) to combat powerful attacks aiming to modify content without detection or to discredit the intrusion detection system with false alarms. In this work, we combine these previous defenses with new ones to defend against attrition attacks as well.

5.1 Admission Control

The purpose of the admission control defense is to ensure that a peer can control the rate at which it considers poll invitations from others, favoring invitations from those who operate at roughly the same rate as itself and penalizing others. We implement admission control using three mechanisms: rate limitation, first-hand reputation, and effort balancing.

Rate Limitation: Without limits on the rate at which they attempt to service requests, peers can be overwhelmed by floods of ostensibly valid requests. *Rate Limitation* suggests that peers should initiate and satisfy requests *no faster than necessary* rather than *as fast as possible*. Because readers access only their local LOCKSS peer, the audit and repair protocol is not subject to end-users' unpredictable request patterns. The protocol can proceed at its own pace, providing an interesting test case for rate limitation.

We identify three possible attacks based on deviation from the *necessary* rate of polling. A *poll rate* adversary seeks to trick victims into either decreasing (e.g., through back-off behavior) or increasing (e.g., through recovery from a failed poll) their rate of calling polls. A *poll flood* adversary seeks, under a multitude of identities, to invite victims into as many frivolous polls as possible to crowd out the legitimate poll requests and thereby reduce the ability of loyal peers to audit and repair their content. A *vote flood* adversary seeks to supply as many bogus votes as possible to exhaust loyal pollers' resources in useless but expensive proofs of invalidity.

Peers defend against all these adversaries by setting their rate limits autonomously, not varying them in response to other peers' actions. Responding to adversity (inquire polls or perceived contention) by calling polls more frequently could aggravate the problem; backing off to a lower rate of polls would achieve the adversary's aim of slowing the detection and repair of damage. Kuzmanovic et al. [26] describe a similar attack in the context of TCP retransmission timers. Because peers do not react, the *poll rate* adversary has no opportunity to attack. The price of this fixed rate of operation is that, absent manual intervention, a peer may take several interpoll intervals to recover from a catastrophic storage failure.

The *poll flood* adversary tries to get victims to over-commit their resources or at least to commit excessively to the adversary. To prevent over-commitment, peers maintain a task schedule of their promises to perform effort, both to generate votes for others and to call their own polls. If the effort of computing the vote solicited by an incoming Poll message cannot be accommodated in the schedule, the invitation is refused. Furthermore, peers limit the rate at which they even *consider* poll invitations (i.e., establishing a secure session, checking their schedule, etc.). A peer sets this rate limit for considering poll invitations according to the rate of poll invitations it sends out to others; this is essentially a *self-clocking* mechanism. We explain how this rate limit is enforced in the first-hand reputation description below. We evaluate our defenses against poll flood strategies in Section 7.3.

The *vote flood* adversary is hamstrung by the fact that votes can be supplied only in response to an invitation by the putative victim poller, and pollers solicit votes at

a fixed rate. Unsolicited votes are ignored.

First-hand reputation: A peer locally maintains and uses first-hand reputation (i.e., history) for other peers. For each AU it preserves, each peer *P* maintains a *known-peers* list containing an entry for every peer *Q* that *P* has encountered in the past, tracking *P*'s exchange of votes with *Q*. The entry holds a reputation grade for *Q*, which takes one of three values: *debt*, *even*, or *credit*. A debt grade means that *Q* has supplied *P* with fewer votes than *P* has supplied *Q*. A credit grade means *P* has supplied *Q* with fewer votes than *Q* has supplied *P*. An even grade means that *P* and *Q* are even in their recent exchanges of votes. Entries in the known-peers list "decay" with time toward the *debt* grade.

In a protocol interaction, the poller and a voter each modify the grade assigned to the other depending on their respective behaviors. If the voter supplies a valid vote and valid repairs for any blocks the poller requests, then the poller increases the grade it assigns to the voter (from debt to even, from even to credit, or from credit to credit) and the voter correspondingly decreases the grade it assigns to the poller. If either the poller or the voter misbehave (e.g., the voter commits to supplying a vote but does not, or the poller does not send a valid evaluation receipt), then the other peer decreases its grade to debt. This is similar to the reciprocative strategy of Feldman et al. [17], in that it penalizes peers who do not reciprocate. This reputation system thus reduces free-riding, as it is not possible for a peer to maintain an even or credit grade without providing valid votes.

Peers randomly drop some poll invitations arriving from previously unknown peers and from known pollers with a debt grade. To discourage identity whitewashing the drop probability imposed on unknown pollers is higher than that imposed on known indebted pollers. Invitations from known pollers with an even or credit grade are not dropped.

Invitations from unknown or indebted pollers are subject to a rigid rate limit; after it admits one such invitation for consideration, a voter enters a *refractory* period. Like the known-peers list, refractory periods are maintained on a per AU basis. During a refractory period, a voter automatically rejects all invitations from unknown or indebted pollers. Consequently, during every refractory period, a voter admits at most one invitation from unknown or indebted peers, plus at most one invitation from each of its fellow peers with a credit or even grade.

Since credit and even grades decay with time, the total "liability" of a peer in the number of invitations it can admit per refractory period is limited to a small constant number. The duration of the refractory period is thus inversely proportional to the rate limit imposed by the peer on the per AU poll invitations it considers.

If a victim peer's clock could be sped up over several

poll intervals then the refractory period could be shortened, increasing the effectiveness of poll flood attacks. The victim would call polls at a faster rate, indebting the victim to its peers and making its invitations less likely to be accepted. However, halving the refractory period from 24 to 12 hours has little effect (see Section 7.4). Doubling the rate of issuing invitations does not affect other peers significantly since the invitations are not accepted. Further, an attack via the Network Time Protocol [32] that doubles a victim's clock rate for months on end would be easy to detect.

Continuous triggering of the refractory period can stop a victim voter from accepting invitations from unknown peers who are loyal; this can limit the choices of voters a poller has to peers that know the poller already. To reduce this impediment to diversity, we institute the concept of peer *introductions*. A peer may introduce to others those peers it considers loyal; peers introduced this way bypass random drops and refractory periods. Introductions are bundled along with nominations during the regular discovery process (Section 4.2). Specifically, a poller randomly partitions the peer identities in a Vote message into outer circle nominations and introductions. A poll invitation from an introduced peer is treated as if coming from a known peer with an even grade. This unobstructed admission consumes the introduction such that at most one introduction is honored per (validly voting) introducer, and unused introductions do not accumulate. Specifically, when consuming the introduction of peer *B* by peer *A* for AU *X*, all other introductions of other introducees by peer *A* for AU *X* are “forgotten,” as are all introductions of peer *B* for *X* by other introducers. Furthermore, introductions by peers who have entered and left the reference list are also removed, and the maximum number of outstanding introductions is capped.

Effort Balancing: If a peer expends more effort to react to a protocol message than did the sender of that message to generate and transmit it, then an attrition attack need consist only of a flow of ostensibly valid protocol messages, enough to exhaust the victim peer's resources.

Real-world attackers may be very powerful but their resources are finite; markets have arisen to allocate pools of compromised machines to competing uses [19]. Raising the computational cost of attacking one target system both absolutely and relative to others will reduce the frequency of attacks. Our simulations are conservative; the unconstrained adversary has ample power for any attack. But our design is more realistic. It adapts the ideas of pricing via processing [15] to discourage attacks from resource-constrained adversaries by *effort balancing* our protocol. We inflate the cost of a request by requiring it to include a proof of computational effort sufficient to ensure that the total cost of generating the request ex-

ceeds that imposed on the receiver both for verifying the effort proof and for satisfying the request. We favor Memory-Bound Functions (MBF) [14] rather than CPU-bound schemes such as “client puzzles” [12] for this purpose, because the spread in memory system performance is smaller than that of CPU performance [13].

Applying an effort filter at each step of a multi-step protocol defends against three attack patterns: first, *desertion* strategies in which the attacker stops taking part some way through the protocol, having spent less effort in the process than the effort inflicted upon his victim; second, *reservation* strategies that cause the victim to commit resources the attacker does not use, making those resources unavailable to other, useful tasks; and, third, *wasteful* strategies in which service is obtained but the result is not “consumed” by the requester as expected by the protocol, in an attempt to minimize the attacker's total expended effort.

Pollers could mount a desertion attack by cheaply soliciting an expensive vote. To discourage this, the poller must include provable effort in its vote solicitation messages (Poll and PollProof) that in total exceeds, by at least an amount described in the next paragraph, the effort required by the voter to verify that effort and to produce the requested vote. Producing a vote amounts to fetching an AU replica from disk, hashing it, and shipping back to the poller one hash per block in the Vote message.

Voters could mount a desertion attack by cheaply generating a bogus vote in response to an expensive solicitation, returning garbage instead of block hashes to waste not merely the poller's solicitation effort but also its effort to verify the hashes. Because the poller evaluates the vote one block at a time, it costs the effort of hashing one block to detect that the vote disagrees with its own AU replica, which may mean either that the vote is bogus, or that the poller's and voter's replicas of the AU differ in that block. The voter must therefore include in the Vote message provable effort sufficient to cover the cost of hashing a single block and of verifying this effort. The extra effort in the solicitation messages referred to above is required to cover the generation of this provable effort.

Pollers could mount a reservation attack by sending a valid Poll message to cause a voter to reserve time for computing a vote in anticipation of a PollProof message the poller never sends. When a voter accepts a poller's invitation, it reserves a block of time in the future to compute the vote. When it is time to begin voting, the voter sets a timeout and waits for the poller to send a PollProof if it has not done so already. If the timeout expires, the voter can reschedule the remainder of the block of time as it pleases. The attack exploits the voter's inability to reallocate the timeout period to another operation by asking for a vote and then never sending a PollProof. To discourage this, pollers must include sufficient *introductory*

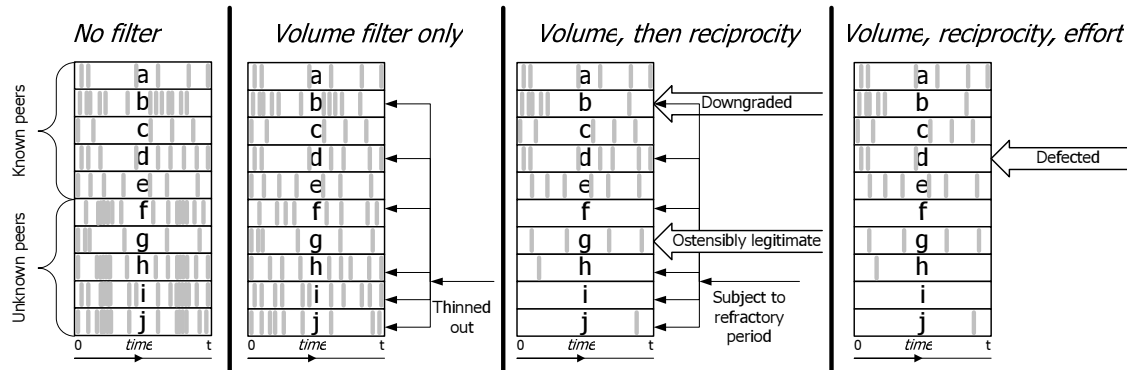


Figure 2: The effects of the logical filters on the incoming stream of poll invitations at a single peer. Rectangles represent poll invitation streams from the different peers *a*, *b*, *c*, etc., during the same time interval $[0, t]$. We show the streams as shaped by the combination of filters, adding one filter at a time, to illustrate each filter’s incremental effect. Within a peer’s poll invitation stream, vertical gray bands represent individual invitation requests.

effort in Poll messages to match the opportunity cost the voter experienced while waiting for the timeout.

Pollers could mount a wasteful attack by soliciting expensive votes and then discarding them unevaluated. To discourage this we require the poller, after evaluating a vote, to supply the voter with an unforgeable *evaluation receipt* proving that it evaluated the vote. Voters generate votes and pollers evaluate them using very similar processes: generating or validating effort proofs and hashing blocks of the local AU replica. Conveniently, generating a proof of effort using our chosen MBF mechanism also generates 160 bits of unforgeable byproduct. The voter remembers the byproduct; the poller uses it as the evaluation receipt to send to the voter. If the receipt matches the voter’s remembered byproduct the voter knows the poller performed the necessary effort, regardless of whether the poller was loyal or malicious.

Section 7.4 shows how effort balancing fares against all three types of attacks mounted by pollers. We omit the evaluation of these attacks by voters, since they are rendered ineffective by the rate limits described above.

The Filters Revisited: Figure 2 illustrates how the defenses of rate limitation, first-hand reputation, and effort balancing, enforced as serial filters over incoming traffic (see Section 3.3), can protect LOCKSS peers from attrition attackers. Among the peers with an initially good standing (*a* through *e*), *a* and *c* maintain a steady balance of requested votes throughout the time interval 0 to *t*. Note that *a* asks for two votes in close succession; this is an instance of a peer expending its “credit.” In contrast, *b* requests many more votes in close succession than justified by its grade and is downgraded to the debt grade by the reciprocity filter, eventually becoming subject to the refractory period. *d* behaves with ostensible legitimacy with regards to the rate of invitations it sends, but misbehaves by deserting (e.g., by not supplying correct effort

proofs) and, as a result, is downgraded to the debt grade by the effort filter. *d*’s subsequent invitations are subject to the refractory period. *g* is initially unknown and therefore subject to the refractory period, but behaves ostensibly legitimately and is upgraded to even or credit grade, freeing itself from the refractory period. Peers *f*, *h*, *i*, and *j* request many more votes than reasonable and occasionally send simultaneous traffic spikes which exceed link capacity; they are thinned out by the volume filter along with other peers’ traffic. These peers, as well as misbehaving peers *b* and *d*, share the same refractory period and therefore only one invitation from them can be accepted per refractory period.

5.2 Desynchronization

The *desynchronization* defense avoids the kind of inadvertent synchronization observed in many distributed systems, typically by randomization. Examples include TCP sender windows at bottleneck routers, clients waiting for a busy server, and periodic routing messages [18]. Peer-to-peer systems in which a peer requesting service must find many others simultaneously available to supply that service (e.g., in a read-one-write-many fault-tolerant system [28]) may encounter this problem. If they do, even absent an attack, moderate levels of peer busyness can prevent the system from delivering services. In this situation, a poll flood attacker may only need to increase peer busyness slightly to have a large effect.

Simulations of poll flood attacks on an earlier version of the protocol [29] showed this effect. Loyal pollers suffered because they needed to find a quorum of voters who could simultaneously vote on an AU. They had to be chosen at random to make directed subversion hard for the adversary. They also needed to have free resources at the specified time, in the face of resource contention

from other peers competing for voters on the same or other AUs. Malign peers had no such constraints, and could invite victims one by one into futile polls.

Peers avoid this problem by soliciting votes individually rather than synchronously, extending the period during which a quorum of votes can be collected before they are all evaluated. A poll is thus a sequence of two-party interactions rather than a single multi-party interaction.

5.3 Redundancy

If the survival of, or access to, an AU relied only on a few replicas, an attrition attack could focus on those replicas, cutting off the communication between them needed for audit and repair. Each LOCKSS peer preserving an AU maintains its own replica and serves it only to its local clients. This massive redundancy helps resist attacks in two ways. First, it ensures that a successful attrition attack must target most of the replicas, typically a large number of peers. Second, it forces the attrition attack to suppress the communication or activity of the targeted peers continuously for a long period. Unless the attack does both, the targeted peers recover by auditing and repairing themselves from the untargeted peers, as shown in Section 7.2. This is because massive redundancy allows peers at each poll to choose a sample of their reference list that is bigger than the quorum and continue to solicit votes from them at random times for the entire duration of a poll (typically 3 months) until the voters accept. Further, the margin between the rate at which peers call polls and the rate at which they suffer undetected damage provides redundancy in time. A single failed poll has little effect on the safety of its caller's replica.

6 Simulation

In this section we give details about the simulation environment and the metrics we use to evaluate the system's effectiveness in meeting its goals.

6.1 Evaluation Metrics

We measure the effectiveness of our defenses against the attrition adversary using four metrics:

Access failure probability: To measure the success of an attrition adversary at increasing the probability that a reader obtains a damaged AU replica, we compute the access failure probability as the fraction of all replicas in the system that are damaged, averaged over all time points in the experiment.

Delay ratio: To measure the degradation an attrition adversary achieves, we compute the delay ratio as the mean time between successful polls at loyal peers with

the system under attack divided by the same measurement without the attack.

Coefficient of friction: To measure the cost of an attack to loyal peers, we measure the coefficient of friction, defined as the average effort expended by loyal peers per successful poll during an attack divided by their average per-poll effort absent an attack.

Cost ratio: To compare the cost of an effortful attack to the adversary and to the defenders, we compute the cost ratio, which is the ratio of the total effort expended by the attackers during an attack to that of the defenders.

6.2 Environment and Adversaries

We run our experiments using Narses [20], a discrete-event simulator that provides facilities for modeling computationally expensive operations, such as computing MBF efforts and hashing documents. Narses allows experimenters to pick from a range of network models that trade off speed for accuracy. A simplistic network model that accounts for network delays but not congestion, except for the side-effects of a pipe stoppage adversary's artificial congestion, suffices for our current focus on application-level effects. Peers' link bandwidths are uniformly distributed among three choices: 1.5, 10, and 100 Mbps, and latencies are uniformly distributed between 1 and 30 ms.

Nodes in the system are divided into two categories: loyal peers and the adversary's minions. Loyal peers are uncompromised peers that execute the protocol correctly. Adversary minions are nodes that collaborate to execute the adversary's attack strategy.

We conservatively simulate the adversary as a cluster of nodes with as many IP addresses and as much compute power as he needs. Each adversary minion has complete and instantaneous knowledge of all adversary state and has a magically incorruptible copy of all AUs. Other assumptions about our adversary that are less relevant to attrition can be found in [30].

To distill the cost of an attack from other efforts the adversary might have to shoulder (e.g., to masquerade as a loyal peer), in these experiments he is completely outside of the network of loyal peers. Loyal peers never ask his minions to vote in polls and he only asks loyal peers to vote in his polls. This differs from LOCKSS adversaries we have studied before [30].

6.3 Simulation Parameters

We evaluate the preservation of a collection of AUs distributed among a population of loyal peers. For simplicity in this stage of our exploration, we assume that each AU is 0.5 GBytes (a large AU in practice). Each peer maintains 50 to 600 AUs. All peers have replicas of all

AUs; we do not yet simulate the diversity of local collections we expect will evolve over time. These simplifications allow us to focus our attention on the common performance of our attrition resistance machinery, ignoring for the time being how that performance varies when AUs vary in size and popularity. Note that our 600 simulated AUs total about 10% of the size of the annual AU intake of a large journal collection such as that of Stanford University Libraries. Adding the equivalent of 10 of today's low-cost PCs per year and consolidating them as old PCs are rendered obsolete is an affordable deployment scenario for such a library. We set all costs of primitive operations (hashing, encryption, L1 cache and RAM accesses, etc.) to match the capabilities of a low-cost PC.

All simulations have a constant loyal peer population of 100 nodes and run for 2 simulated years, with 3 runs per data point. Each peer runs a poll on each of its AUs on average every 3 months. Each poll uses a quorum of 10 peers and considers landslide agreement as having a maximum of 3 disagreeing votes. These parameters were empirically determined from previous iterations of the deployed beta protocol. We set the fixed drop probability to be 0.90 for unknown peers and 0.80 for indebted peers.

We set the fixed drop probability for indebted peers and the cost of verifying an introductory effort so that the cumulative introductory effort expended by an effortful attack on dropped invitations is more than the voter's effort to consider the adversary's eventually *admitted* invitation. Since an adversary has to try with indebted identities on average 5 times to be admitted (thanks to the $1 - 0.8 = 0.2$ admission probability), we set the introductory effort to be 20% of the total effort required of a poller; by the time the adversary has gotten his poll invitation admitted, even if he defects for the rest of the poll, he has already expended on average 100% of the effort he would have, had he behaved well in the first place.

Memory limits in the Java Virtual Machine prevent Narses from simulating more than about 50 AUs/peer in a single run. We simulate 600-AU collections by *layering* 50 AUs/peer runs, adding the tasks caused by one layer's 50 AUs to the task schedule for each peer accumulated during the preceding layers. In effect, layer n is a simulation of 50 AUs on peers already running a realistic workload of $50(n - 1)$ AUs. The effect is to over-estimate the peer's busyness for AUs in higher layers and under-estimate it for AUs in lower layers; AUs in a layer compete for the resources left over by lower layers, but AUs in lower layers are unaffected by the resources used in higher layers. We have validated this technique against unlayered simulations in smaller collections, as well as against simulations in which inflated per-AU preservation costs cause similar levels of peer load; we found negligible differences.

We are currently exploring the parameter space but

use the following heuristics to help determine parameter values. The refractory period of one day allows for 90 invitations from unknown or indebted peers to be accepted per 90-day interpoll interval; in contrast, a peer requires an average of 30 votes per poll and, because of self-clocking, should be able to accept at least an average of 30 poll invitations per interpoll interval. Consequently, the one-day refractory period allows up to a total of 120 invitations per poll period, four times the rate of poll invitations that should be expected in the absence of attacks.

7 Results

The probability of access failure summarizes the success of an attrition attack. We start by establishing a baseline rate of access failures absent an attack. We then assess the effectiveness against this baseline of the effortless attacks we consider: network-level flooding attacks on the volume filter in Section 7.2, and Sybil attacks on the reciprocity filter in Section 7.3. Finally, in Section 7.4 we assess against this baseline each of the effortful attacks corresponding to each effort filter.

In each case we show the effect of increasing scales of attack on the access failure probability, and relevant supporting graphs including the delay ratio, the coefficient of friction, and for effortful attacks the cost ratio.

Our mechanisms for defending against an attrition adversary raise the effort required per loyal peer. To achieve a bound on access failure probabilities, one must be willing to over-provision the system to accommodate the extra effort. Over-provisioning the system by a constant factor defends it against application-level attrition attacks of unlimited power (Sections 7.3 and 7.4).

7.1 Baseline

The LOCKSS polling process is intended to detect and recover from storage damage that is *not* detected locally, from causes such as "bit rot," human error and attack. Our simulated peers suffer such damage at rates of one block in 1 to 5 disk years (50 AUs per disk). This is an aggressively inflated rate of undetected damage, given that, for instance, it is 125-400% the rate of *detected* failures in Talagala's study of IDE drives in a large disk farm [45]. Experience with the IDE drives in deployed LOCKSS peers covers about 10 times as many disk years but with less reliable data collection; it suggests much lower detected failure rates.

Figure 3 plots access failure probability versus the interpoll interval. It shows that as the interpoll interval increases relative to the mean interval between storage failures, access failure probability increases because damage

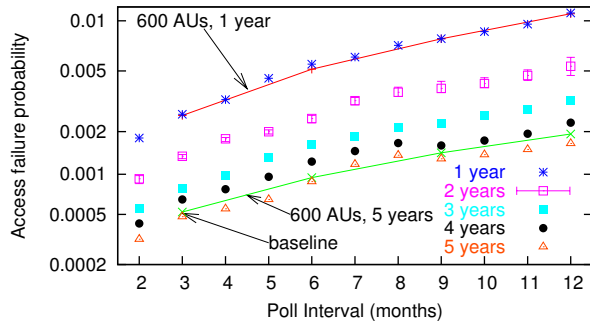


Figure 3: Mean access failure probability (y axis in log scale) for increasing interpoll intervals (x axis) at variable mean times between storage failure (from 1 to 5 years per disk), absent an attack. We show results for collection sizes of 50 AUs (points only) and of 600 AUs (lines and points). We show minimum and maximum values for the 2-year data set; this variance is representative of all measurements, which we omit for clarity.

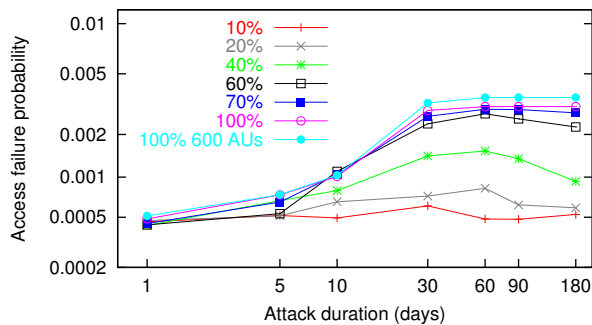


Figure 4: The access failure probability (y axis in log scale) observed during repeated pipe stoppage attacks of varying duration (x axis in log scale), covering between 10 and 100% of the peers.

takes longer to detect and repair. The access failure probability is similar for a 50-AU collection all the way up to a 600-AU collection (we omit intermediate collection sizes for clarity).

For comparison purposes in the rest of the experiments, the baseline access failure probability of 4.8×10^{-4} for a 50-AU collection and of 5.2×10^{-4} for a 600-AU collection correspond to our interpoll interval of 3 months and a storage damage rate of one block per 5 disk years. With these parameters, a machine preserving 600 AUs has an average load of 9%, and a machine preserving 50 AUs has a 0.7% average load.

7.2 Targeting the Volume Filter

The “pipe stoppage” adversary models packet flooding and more sophisticated attacks [26]. This adversary suppresses all communication between some proportion of the total peer population (its *coverage*) and other LOCKSS peers. During a pipe-stoppage attack, local readers may still access content. The adversary subjects a victim to a period of pipe stoppage lasting between 1 and 180 days. Each attack is followed by a 30-day recuperation period, during which communication is restored to the victim; this pattern is repeated for the entire experiment. To lower the probability that a recuperating peer can contact another peer, the adversary schedules his attacks such that there is little overlap in peers’ recuperation periods. We performed experiments with an adversary that schedules his attacks so that all victims’ recuperation periods completely overlap, but found that the low-overlap adversary caused more damage, so we present results from the low-overlap adversary.

Figure 4 plots the access failure probability versus the attack duration for varying coverage values (10 to 100%). As expected, the access failure probability increases as the coverage of the attack increases, though the attack covering 70% of the peer population is almost as effective as the 100% attack. In the extreme, the 180 day attack over 100% of the 600-AU collection raises the access failure probability to 3.5×10^{-3} ; this is within tolerable limits for services open to the Internet.

For attacks between 20% and 60% coverage, the access failure probability peaks at an attack duration of 60 days and decreases for larger durations. The 180 day attack is less damaging for these coverage values because, while the adversary focuses on a smaller number of peers for a longer time, the rest of the peers continue polling. The 30 to 60 day attacks cycle across more victims and interrupt more polls, wasting peers’ time and tarnishing their reputations, while 1 to 10 day attacks are too short to interrupt many polls. As the attack coverage grows from 70%, the 180 day attack disables such a significant portion of the network that the peers free of attack have great difficulty finding available peers and the access failure probability increases beyond the 60 day attack.

Figures 5 and 6 plot the delay ratio and coefficient of friction, respectively, versus attack duration. We find that attacks must last longer than 30 days to raise the delay ratio by an order of magnitude. Similarly, the coefficient of friction during repeated attacks that last less than a few days each is negligibly greater than 1. For very long attacks that completely shut down the victim’s Internet, the coefficient can reach 6700, making pipe stoppage the most cost-effective strategy for the attrition adversary.

As attack durations grow to 30 days and beyond, the adversary succeeds in decreasing the total number of suc-

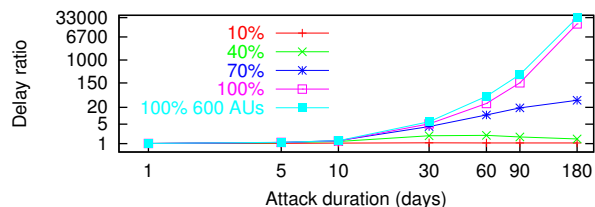


Figure 5: The delay ratio (y axis in log scale) imposed by repeated pipe stoppage attacks of varying duration (x axis in log scale) and coverage of the population. Absent an attack, this metric has value 1.

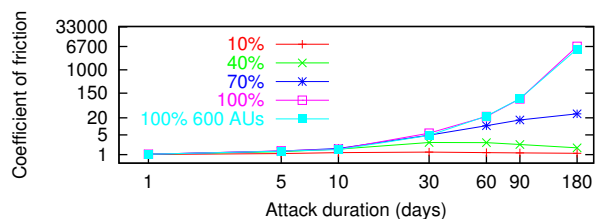


Figure 6: The coefficient of friction (y axis in log scale) imposed by pipe stoppage attacks of varying duration (x axis in log scale) and coverage of the population.

cessful polls. For example, attacks against 100% of the population with a 30 day duration reduce the number of successful polls to 1/5 the number absent attack. However, the average machine load during recuperation remains within 2 to 3 times the baseline load — a result of designing the protocol to limit increases in resource consumption while under attack. Fewer successful polls and nearly constant resource consumption for increasing attack durations drives up the average cost of a successful poll, and with it the coefficient of friction.

7.3 Targeting the Reciprocity Filter

The reciprocity adversary attacks our admission control defenses aiming to reduce the likelihood of a victim admitting a loyal poll request by triggering that victim's refractory period as often as possible. This adversary sends cheap garbage invitations to varying fractions of the peer population for varying periods of time separated by a fixed recuperation period of 30 days. The adversary sends invitations using poller addresses unknown to the victims. These, when eventually admitted, cause those victims to enter their refractory periods and drop all subsequent invitations from unknown and indebted peers.

Figure 7 shows that these attacks have little effect. The access failure probability is raised to 5.9×10^{-4} when the duration of the attack reaches the entire duration of our simulations (2 years) for full population coverage and a 600-AU collection. At that attack intensity, loyal

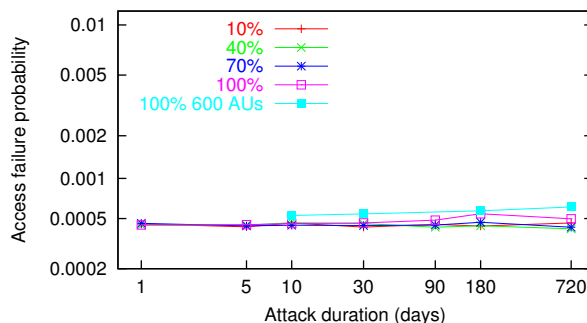


Figure 7: The access failure probability (y axis in log scale) for attacks of increasing duration (x axis in log scale) by the admission control adversary over 10 to 100% of the peer population. The scale and size of the graph match Figures 3 and 4 to facilitate comparison.

peers no longer admit poll invitations from unknown or indebted loyal peers, unless supported by an introduction. This causes discovery to operate more slowly; loyal peers waste their resources on introductory effort proofs that are summarily rejected by peers in their refractory period. This wasted effort, when sustained over years, raises the coefficient of friction by 33%, (much less than the friction caused by pipe stoppage), and raises average machine load from 9% to 11%. The delay ratio is largely unaffected by this adversary. Consequently, the first effect of this adversary, increasing load in loyal peers, is tolerable given a practical level of over-provisioning.

We switch our attention to the other effect of this adversary, namely, the suppression of invitations from unknown or indebted peers, which introductions are intended to mitigate. We have repeated the experiments with 600 AUs, in which the adversary attacks 100% of the peer population, with introductions disabled. Without introductions, the shorter attacks cause a higher coefficient of friction, much closer to pipe stoppage attacks, whereas longer attacks are largely unaffected. For comparison, suppressing introductions for attack durations of 10 days raises the coefficient of friction from 1.03 to 1.16, vs. 1.51 for pipe stoppage; in contrast, suppressed introductions for attack durations of six months raises the coefficient of friction from 1.34 to 1.36, vs. 6700 for pipe stoppage. The absence of introductions does not make this attack markedly worse in terms of load increase.

The major consequence of unknown and indebted invitation suppression without introductions is that victims call polls almost exclusively composed of voters from their friends list, who are more likely to accept a poll invitation from a fellow friend. This reliance increases as the attack lasts longer. It is undesirable because it allows an adversary to predict closely the membership of a poll (mostly the poller's friends), promoting focused poll dis-

Defection	Coeff. friction	Cost ratio	Delay ratio	Access failure
INTRO	1.40	1.93	1.11	4.99×10^{-4}
	1.31	2.04	1.10	6.35×10^{-4}
REMAIN- ING	2.61	1.55	1.11	5.90×10^{-4}
	2.50	1.60	1.10	6.16×10^{-4}
NONE	2.60	1.02	1.11	5.58×10^{-4}
	2.49	1.06	1.10	6.19×10^{-4}

Table 1: The effect of the brute force adversary defecting at various points in the protocol on the coefficient of friction, the cost ratio, the delay ratio, and the access failure probability. For each point, the upper numbers correspond to the 50-AU collection and the lower numbers correspond to the 600-AU collection.

ruptions. The main function of introductions is thus to ensure the unpredictability of poll memberships.

Note that techniques such as blacklisting, commonly used to defeat denial-of-service attacks in the context of email spam, or server selection [17] by which pollers only invite voters they believe will accept, could significantly reduce the friction caused by the admission control attack. However, we have yet to explore whether these defenses are compatible with our goal of protecting against subversion attacks that operate by biasing the opinion poll sample toward corrupted peers [30].

7.4 Targeting the Effort Filters

To attack filters downstream of the reciprocity filter, the adversary must get through as fast as possible. We consider an attack by a “brute force” adversary who continuously sends enough poll invitations with valid introductory efforts to get past the random drops; such invitations cannot arrive from credit or even identities at the steady attack state, because they are more frequent than what is considered legitimate. Since unknown peers suffer more random drops than peers in debt, the adversary launches attacks from indebted addresses. We conservatively initialize all adversary addresses with a debt grade at all loyal peers. We also give the adversary an oracle that allows him to inspect all the loyal peers’ schedules. This avoids his wasting introductory efforts due to scheduling conflicts at loyal peers.

Once through the reciprocity filter, the adversary can defect at any stage of the protocol exchange: after providing the introductory effort in the Poll message (INTRO) by never following up with a PollProof, after providing the remaining effort in the PollProof message (REMAINING) by never following up with an EvaluationReceipt, or not defecting at all (NONE).

Table 1 shows that the brute force adversary’s most

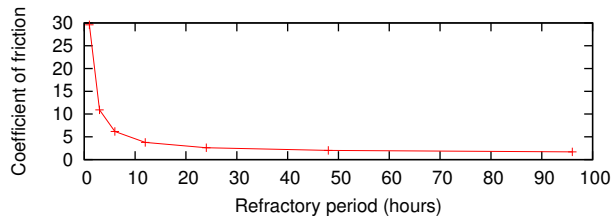


Figure 8: Coefficient of friction during brute force attacks against 50 AUs. The refractory period varies from 1 to 96 hours.

cost-effective strategy (i.e., with the lowest cost ratio metric) is to participate fully in the protocol; doing so he can raise loyal peers’ preservation cost (i.e., their coefficient of friction) to a factor of 2.60 (2.49 for the large collection, which equates to an average machine load of 21%). To defend against this increase in cost, LOCKSS peers must over-provision their resources by a reasonable amount. The baseline probability of access failure rises to 6.19×10^{-4} at a cost almost identical to that incurred by the defenders (a cost ratio of 1.06). Fortunately, this continuous attack even from a brute force adversary unconcerned by his own effort expenditure is unable to increase the access failure probability of the victims greatly; the rate limits prevent him from bringing his advantage in resources to bear. Similar behavior in earlier work [30] prevents a different unconstrained adversary from stealthily modifying content.

We measured the effectiveness of the refractory period in rate limiting poll flood attacks against the brute force adversary that does not defect, since this strategy has the best cost/benefit ratio among the brute force strategies. Figure 8 shows the coefficient of friction during a brute force attack on 50 AUs where the refractory period varies from 1 to 96 hours. With a shorter refractory period, poll invitations from the attacker are accepted by the victims at a greater rate, driving up the coefficient of friction. With the refractory period at one hour, the average machine load at the victim peers is 21%. If only 50 AUs consume 21% of a peer’s processing time, an average peer cannot support 600 AUs while under attack. With the refractory period of 24 hours, the peers’ average load supporting 50 AUs is only 2%.

On the other hand, the graph shows that lengthening the refractory period beyond 24 hours would not greatly reduce the coefficient of friction. Furthermore, increasing the refractory period decreases the probability of a peer accepting legitimate poll invitations from unknown or indebted peers, since voters accept fewer of these invitations per unit time. A very long refractory period stifles the discovery process of pollers finding new voters and causes increased reliance on a poller’s friends list. Similar behavior occurs when introductions are removed

from the protocol (see Section 7.3).

Thus a shorter refractory period increases the probability of voters accepting invitations from legitimate, unknown pollers, but it also increases damage during a poll flood attack. Our choice of 24 hours limits the harm an attacker can do while accepting enough legitimate poll invitations from unknown or indebted peers for the discovery process to function.

In the analysis above, we conservatively assume that the brute force adversary uses attacking identities in the debt grade of their victims. Space constraints lead us to omit experiments with an adversary whose minions may be in either even or credit grade. This adversary polls a victim only after he has supplied that victim with a vote, then defects in any of the ways described above. He then recovers his grade at the victim by supplying an appropriate number of valid votes in succession. Each vote he supplies is used to introduce new minions that thereby bypass the victim's admission control before defecting. This attack requires the victim to invite minions into polls and is sufficiently rate-limited to be less effective than brute force. It is further limited by the decay of first-hand reputation toward the debt grade. We leave the details for an extended version of this paper.

8 Related Work

In this section we first describe the most significant ways in which the new LOCKSS protocol differs from our previous efforts. We then list work that describes the nature and types of denial of service attacks, as well as related work that applies defenses similar to ours.

The protocol described here is derived from earlier work [30] in which we covered the background of the LOCKSS system. That protocol used redundancy, rate limitation, effort balancing, bimodal behavior (polls must be won or lost by a landslide) and friend bias (soliciting some percentage of votes from peers on the friends list) to prevent powerful adversaries from modifying the content without detection, or discrediting the system with false alarms. In this work, we target the protocol's vulnerability to attrition attacks by reinforcing our previous defenses with admission control, desynchronization, and redundancy.

Another major difference between our prior work and the protocol described in this paper is our treatment of repair. In the previous protocol voting and repair were separated into two phases. When pollers determined repair was necessary, they requested a complete copy of the document from the publisher, if still available, or from a peer for whom they had previously supplied votes. This had at least three problems. First, pollers requested repairs only when needed, signaling the vulnerability of those pollers' content to an adversary. Second, the repair

mechanism was only exercised when content recovery was needed. Mechanisms exercised only during emergencies are unlikely to work [35]. Finally, this left the system more vulnerable to free-riding, since a peer could supply votes but later defect when the poller requested a costly repair. We address all three problems through restructuring the repair mechanism (as described in Section 4.3) to integrate block repairs, including "frivolous" repairs, into the actual evaluation of votes.

A third significant difference in the protocol supports our desynchronization defense. In the previous protocol, loyal pollers needed to find a quorum of voters who could simultaneously vote on an AU. Instead, the poller now solicits and obtains votes one at a time, across the duration of a poll, and only evaluates the outcome of a poll once it has accumulated all requisite votes.

Our attrition adversary draws on a wide range of work in detecting [23], measuring [33], and combating [2, 27, 41, 42] network-level DDoS attacks capable of stopping traffic to and from our peers. This work observes that current attacks are not simultaneously of high intensity, long duration, and high coverage (many peers) [33].

Redundancy is a key to survival during some DoS attacks, because pipe stoppage appears to other peers as a failed peer. Many systems use redundancy to mask storage failure [25]. Byzantine Fault Tolerance [7] is related to the LOCKSS opinion polling mechanism in its goal of managing replicas in the face of attack. It provides stronger guarantees but requires that no more than one third of the replicas are faulty or misbehaving. In a distributed system, such as the LOCKSS system, that is spread across the Internet, we cannot assume an upper bound on the number of misbehaving peers. We therefore aim for system performance to degrade gracefully with increasing numbers of misbehaving peers, rather than fail suddenly when a critical threshold is reached. Routing along multiple redundant paths in Distributed Hash Tables (DHTs) has been suggested as a way of increasing the probability that a message arrives at its intended recipient despite nodes dropping messages due to malice [6] or pipe stoppage [24].

Rate limits are effective in slowing the spread of viruses [43, 48]. They have also been suggested for limiting the rate at which peers can join a DHT [6, 47] as a defense against attempts to control part of the hash space. Our work suggests that DHTs will need to rate limit not only joins but also stores to defend against attrition attacks. Another study [40] suggests that the increased latency this causes will not affect users' behavior.

Effort balancing is used as a defense against spam, which may be considered an application-level DoS attack and has received the bulk of the attention in this area. Our effort balancing defense draws on pricing via processing concepts [15]. We measure cost by memory

cycles [1, 14]; others use CPU cycles [4, 15] or even Turing tests [44]. Crosby et al. [10] show that worst-case behavior of application algorithms can be exploited in application-level DoS attacks; our use of nonces and the bounded verification time of MBF avoid this risk. In the LOCKSS system we avoid strong peer identities and infrastructure changes, and therefore rule out many techniques for excluding malign peers such as Secure Overlay Services [24].

Related to first-hand reputation is the use of game-theoretic analysis of peer behavior by Feldman et al. [17] to show that a reciprocative strategy in admission control policy can motivate cooperation among selfish peers.

Admission control has been used to improve the usability of overloaded services. For example, Cherkasova et al. [8] propose admission control strategies that help protect long-running Web service sessions (i.e., related sequences of requests) from abrupt termination. Preserving the responsiveness of Web services in the face of demand spikes is critical, whereas LOCKSS peers need only manage their resources to make progress at the necessary rate in the long term. They can treat demand spikes as hostile behavior. In a P2P context, Daswani et al. [11] use admission control (with rate limiting) to mitigate the effects of a query flood attack against superpeers in unstructured file-sharing peer-to-peer networks.

Golle and Mironov [21] provide compliance enforcement in the context of distributed computation using a receipt technique similar to ours. Random auditing using challenges and hashing has been proposed [9, 47] as a means of enforcing trading requirements in some distributed storage systems.

In DHTs waves of synchronized routing updates caused by joins or departures result in instability during periods of high churn. Bamboo's [36] desynchronization defense using lazy updates is effective.

9 Future Work

We have three immediate goals for future work. First, we observe that although the protocol is symmetric, the attrition adversary's use of it is asymmetric. It may be that adaptive behavior of the loyal peers can exploit this asymmetry. For example, loyal peers could modulate the probability of acceptance of a poll request according to their recent busyness. The effect would be to raise the marginal effort required to increase the loyal peer's busyness as the attack effort increases. Second, we need to understand how our defenses against attrition work in a more dynamic environment, where new loyal peers continually join the system over time. Third, we need to consider combined adversary strategies; an adversary could weaken the system with an attrition attack in preparation for some other type of attack.

10 Conclusion

The defenses of this paper equip the LOCKSS system to resist attrition well. First, application-level attrition attacks, even from adversaries with no resource constraints and sustained for two years, can be defeated with reasonable over-provisioning. Such over-provisioning is natural in our application, but further work may significantly reduce the required amount. Second, the strategy that provides an unconstrained adversary with the greatest impact on the system is to behave as a large number of new loyal peers. Third, network-level attacks do not affect the system significantly unless they are (a) intense enough to stop all communication between peers, (b) widespread enough to target all of the peers, and (c) sustained over months.

Digital preservation is an unusual application, in that the goal is to prevent things from happening. The LOCKSS system resists failures and attacks from powerful adversaries *without* normal defenses such as long-term secrets and central administration. The techniques that we have developed may be primarily applicable to preservation, but we hope that our conservative design will assist others in building systems that better meet society's need for more reliable and defensible systems.

Both the LOCKSS project and the Narses simulator are hosted at SourceForge, and both carry BSD-style Open Source licenses. Implementation of this protocol in the production LOCKSS system is in progress.

11 Acknowledgments

We are grateful to Kevin Lai, Joe Hellerstein, Yanto Muliadi, Prashanth Bungale, Geoff Goodell, Ed Swierk, Lucy Cherkasova, and Sorav Bansal for their help. We owe many thanks to our shepherd, Carla Ellis, and the anonymous reviewers for their help in improving this paper. Finally, we are especially thankful to Vicky Reich, the director of the LOCKSS program.

This work is supported by the National Science Foundation (Grant No. 9907296) and by the Andrew W. Mellon Foundation. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of these funding agencies.

References

- [1] ABADI, M., BURROWS, M., MANASSE, M., AND WOBBER, T. Moderately Hard, Memory-bound Functions. In *NDSS* (2003).
- [2] ARGYRAKI, K., AND CHERITON, D. Active Internet Traffic Filtering: Real-time Response to Denial of Service Attacks. In *USENIX* (Apr. 2005).

- [3] ARL – ASSOCIATION OF RESEARCH LIBRARIES. ARL Statistics 2000-01. <http://www.arl.org/stats/arlstat/01pub/intro.html>, 2001.
- [4] BACK, A. Hashcash - a denial of service counter measure, 2002. <http://www.hashcash.org/hashcash.pdf>.
- [5] CANTARA, L. Archiving Electronic Journals. <http://www.diglib.org/preserve/ejp.htm>, 2003.
- [6] CASTRO, M., DRUSCHEL, P., GANESH, A., ROWSTRON, A., AND WALLACH, D. S. Secure Routing for Structured Peer-to-Peer Overlay Networks. In *OSDI* (2002).
- [7] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance. In *OSDI* (1999).
- [8] CHERKASOVA, L., AND PHAAL, P. Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites. *IEEE Trans. on Computers* 51, 6 (2002).
- [9] COX, L. P., AND NOBLE, B. D. Samsara: Honor Among Thieves in Peer-to-Peer Storage. In *SOSP* (2003).
- [10] CROSBY, S., AND WALLACH, D. S. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security Symp.* (2003).
- [11] DASWANI, N., AND GARCIA-MOLINA, H. Query-Flood DoS Attacks in Gnutella. In *ACM Conf. on Computer and Communications Security* (2002).
- [12] DEAN, D., AND STUBBLEFIELD, A. Using Client Puzzles to Protect TLS. In *USENIX Security Symp.* (2001).
- [13] DOUCEUR, J. The Sybil Attack. In *1st Intl. Workshop on Peer-to-Peer Systems* (2002).
- [14] DWORK, C., GOLDBERG, A., AND NAOR, M. On Memory-Bound Functions for Fighting Spam. In *CRYPTO* (2003).
- [15] DWORK, C., AND NAOR, M. Pricing via Processing. In *CRYPTO* (1992).
- [16] ELECTRONIC FRONTIER FOUNDATION. DMCA Archive. <http://www.eff.org/IP/DMCA/>.
- [17] FELDMAN, M., LAI, K., STOICA, I., AND CHUANG, J. Robust Incentive Techniques for Peer-to-Peer Networks. In *ACM Electronic Commerce* (2004).
- [18] FLOYD, S., AND JACOBSON, V. The Synchronization of Periodic Routing Messages. *ACM Trans. on Networking* 2, 2 (1994).
- [19] GEER, D. Malicious Bots Threaten Network Security. *IEEE Computer* 38, 1 (Jan. 2005), 18–20.
- [20] GIULI, T., AND BAKER, M. Narses: A Scalable, Flow-Based Network Simulator. Technical Report arXiv:cs.PF/0211024, CS Department, Stanford University, Nov. 2002.
- [21] GOLLE, P., AND MIRONOV, I. Uncheatable Distributed Computations. *Lecture Notes in Computer Science* 2020 (2001).
- [22] HORLINGS, J. Cd-r's binnen twee jaar onleesbaar. <http://www.pc-active.nl/toonArtikel.asp?artikelID=508>, 2003. <http://www.cdfreaks.com/news/7751>.
- [23] HUSSAIN, A., HEIDEMANN, J., AND PAPADOPOULOS, C. A Framework for Classifying Denial of Service Attacks. In *SIGCOMM* (2003).
- [24] KEROMYTIS, A., MISRA, V., AND RUBENSTEIN, D. SOS: Secure Overlay Services. In *SIGCOMM* (2002).
- [25] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An Architecture for Global-Scale Persistent Storage. In *ASPLOS* (2000).
- [26] KUZMANOVIC, A., AND KNIGHTLY, E. W. Low-Rate TCP-Targeted Denial of Service Attacks (The Shrew vs. the Mice and Elephants). In *SIGCOMM* (2003).
- [27] MAHAJAN, R., BELLOVIN, S., FLOYD, S., IOANNIDIS, J., PAXSON, V., AND SHENKER, S. Controlling high bandwidth aggregates in the network. *Comp. Comm. Review* 32, 3 (2002).
- [28] MALKHI, D., AND REITER, M. Byzantine Quorum Systems. *J. Distributed Computing* 11, 4 (1998), 203–213.
- [29] MANIATIS, P., GIULI, T., ROUSSOPOULOS, M., ROSENTHAL, D., AND BAKER, M. Impeding Attrition Attacks on P2P Systems. In *SIGOPS European Workshop* (2004).
- [30] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T., ROSENTHAL, D. S. H., BAKER, M., AND MULIADI, Y. Preserving Peer Replicas By Rate-Limited Sampled Voting. In *SOSP* (2003).
- [31] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A Fast File System for UNIX. *ACM Trans. on Computer Systems* 2, 3 (1984), 181–197.
- [32] MILLS, D. L. Internet Time Synchronization: The Network Time Protocol. *IEEE Trans. on Communications* 39, 10 (Oct. 1991), 1482–1493.
- [33] MOORE, D., VOELKER, G. M., AND SAVAGE, S. Inferring Internet Denial-of-Service Activity. In *USENIX Security Symp.* (2001).
- [34] NEEDHAM, R. Denial of Service. In *ACM Conf. on Computer and Communications Security* (1993).
- [35] REASON, J. *Human Error*. Cambridge University Press, 1990.
- [36] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a DHT. In *USENIX* (2004).
- [37] RODRIGUES, R., AND LISKOV, B. Byzantine Fault Tolerance in Long-Lived Systems. In *FuDiCo* (2004).
- [38] ROSENTHAL, D. S., ROUSSOPOULOS, M., GIULI, T., MANIATIS, P., AND BAKER, M. Using Hard Disks For Digital Preservation. In *Imaging Sci. and Tech. Archiving Conference* (2004).
- [39] ROSENTHAL, D. S. H., AND REICH, V. Permanent Web Publishing. In *USENIX, Freenix Track* (2000).
- [40] SAROIU, S., GUMMADI, K., DUNN, R., GRIBBLE, S., AND LEVY, H. An Analysis of Internet Content Delivery Systems. In *OSDI* (2002).
- [41] SAVAGE, S., WETHERALL, D., KARLIN, A., AND ANDERSON, T. Practical Network Support for IP Traceback. In *SIGCOMM* (2000).
- [42] SNOEREN, A., PARTRIDGE, C., SANCHEX, L., JONES, C. E., TCHAKOUNTIO, F., KENT, S. T., AND STRAYER, T., W. Hash-based IP Traceback. In *SIGCOMM* (2001).
- [43] SOMAYAJI, A., AND FORREST, S. Automated Response Using System-Call Delays. In *Unix Security Symp.* (2000).
- [44] SPAM ARREST, LLC. Take Control of your Inbox. <http://spamarrest.com>.
- [45] TALAGALA, N. *Characterizing Large Storage Systems: Error Behavior and Performance Benchmarks*. PhD thesis, CS Div., Univ. of California at Berkeley, Berkeley, CA, USA, Oct. 1999.
- [46] TENOPIR, C. Online Scholarly Journals: How Many? *The Library Journal*, 2 (2004). <http://www.libraryjournal.com/index.asp?layout=articlePrint&articleID=C%A374956>.
- [47] WALLACH, D. A Survey of Peer-to-Peer Security Issues. In *Intl. Symp. on Software Security* (2002).
- [48] WILLIAMSON, M. Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. In *Annual Computer Security Applications Conf.* (2002).

Notes

¹LOCKSS is a trademark of Stanford University.

Peer-to-Peer Communication Across Network Address Translators

Bryan Ford
Massachusetts Institute of Technology
baford@mit.edu

Pyda Srisuresh
Caymas Systems, Inc.
srisuresh@yahoo.com

Dan Kegel
dank@kegel.com

*J'fais des trous, des petits trous...
toujours des petits trous*
- S. Gainsbourg

Abstract

Network Address Translation (NAT) causes well-known difficulties for peer-to-peer (P2P) communication, since the peers involved may not be reachable at any globally valid IP address. Several NAT traversal techniques are known, but their documentation is slim, and data about their robustness or relative merits is slimmer. This paper documents and analyzes one of the simplest but most robust and practical NAT traversal techniques, commonly known as hole punching. Hole punching is moderately well-understood for UDP communication, but we show how it can be reliably used to set up peer-to-peer TCP streams as well. After gathering data on the reliability of this technique on a wide variety of deployed NATs, we find that about 82% of the NATs tested support hole punching for UDP, and about 64% support hole punching for TCP streams. As NAT vendors become increasingly conscious of the needs of important P2P applications such as Voice over IP and online gaming protocols, support for hole punching is likely to increase in the future.

1 Introduction

The combined pressures of tremendous growth and massive security challenges have forced the Internet to evolve in ways that make life difficult for many applications. The Internet's original uniform address architecture, in which every node has a globally unique IP address and can communicate directly with every other node, has been replaced with a new *de facto* Internet address architecture, consisting of a global address realm and many private address realms interconnected by Network Address Translators (NAT). In this new address architecture, illustrated in Figure 1, only nodes in the main, global address realm

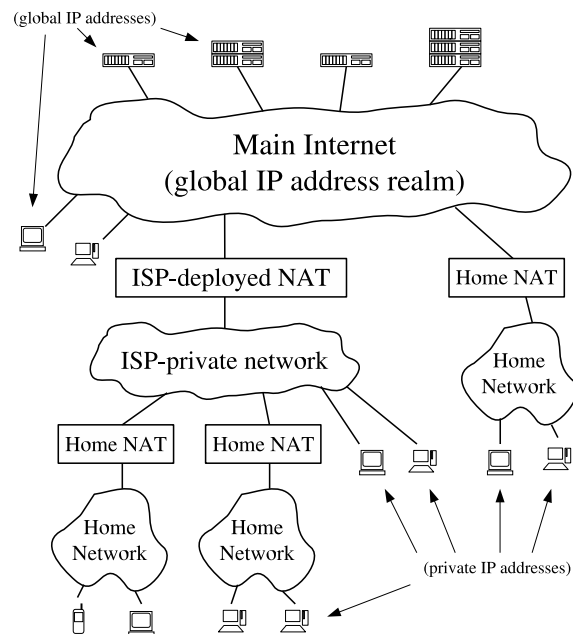


Figure 1: Public and private IP address domains

can be easily contacted from anywhere in the network, because only they have unique, globally routable IP addresses. Nodes on private networks can connect to other nodes on the same private network, and they can usually open TCP or UDP connections to well-known nodes in the global address realm. NATs on the path allocate temporary public endpoints for outgoing connections, and translate the addresses and port numbers in packets comprising those sessions, while generally blocking all incoming traffic unless otherwise specially configured.

The Internet's new *de facto* address architecture is suitable for client/server communication in the typical case when the client is on a private network and the server is in the global address realm. The architecture makes it dif-

cult for two nodes on *different* private networks to contact each other directly, however, which is often important to the “peer-to-peer” communication protocols used in applications such as teleconferencing and online gaming. We clearly need a way to make such protocols function smoothly in the presence of NAT.

One of the most effective methods of establishing peer-to-peer communication between hosts on different private networks is known as “hole punching.” This technique is widely used already in UDP-based applications, but essentially the same technique also works for TCP. Contrary to what its name may suggest, hole punching does not compromise the security of a private network. Instead, hole punching enables applications to function *within* the the default security policy of most NATs, effectively signaling to NATs on the path that peer-to-peer communication sessions are “solicited” and thus should be accepted. This paper documents hole punching for both UDP and TCP, and details the crucial aspects of both application and NAT behavior that make hole punching work.

Unfortunately, no traversal technique works with all existing NATs, because NAT behavior is not standardized. This paper presents some experimental results evaluating hole punching support in current NATs. Our data is derived from results submitted by users throughout the Internet by running our “NAT Check” tool over a wide variety of NATs by different vendors. While the data points were gathered from a “self-selecting” user community and may not be representative of the true distribution of NAT implementations deployed on the Internet, the results are nevertheless generally encouraging.

While evaluating basic hole punching, we also point out variations that can make hole punching work on a wider variety of existing NATs at the cost of greater complexity. Our primary focus, however, is on developing the *simplest* hole punching technique that works cleanly and robustly in the presence of “well-behaved” NATs in any reasonable network topology. We deliberately avoid excessively clever tricks that may increase compatibility with some existing “broken” NATs in the short term, but which only work some of the time and may cause additional unpredictability and network brittleness in the long term.

Although the larger address space of IPv6 [3] may eventually reduce the need for NAT, in the short term IPv6 is *increasing* the demand for NAT, because NAT itself provides the easiest way to achieve interoperability between IPv4 and IPv6 address domains [24]. Further, the anonymity and inaccessibility of hosts on private networks has widely perceived security and privacy benefits. Firewalls are unlikely to go away even when there are enough IP addresses: IPv6 firewalls will still commonly

block unsolicited incoming traffic by default, making hole punching useful even to IPv6 applications.

The rest of this paper is organized as follows. Section 2 introduces basic terminology and NAT traversal concepts. Section 3 details hole punching for UDP, and Section 4 introduces hole punching for TCP. Section 5 summarizes important properties a NAT must have in order to enable hole punching. Section 6 presents our experimental results on hole punching support in popular NATs, Section 7 discusses related work, and Section 8 concludes.

2 General Concepts

This section introduces basic NAT terminology used throughout the paper, and then outlines general NAT traversal techniques that apply equally to TCP and UDP.

2.1 NAT Terminology

This paper adopts the NAT terminology and taxonomy defined in RFC 2663 [21], as well as additional terms defined more recently in RFC 3489 [19].

Of particular importance is the notion of session. A *session endpoint* for TCP or UDP is an (IP address, port number) pair, and a particular *session* is uniquely identified by its two session endpoints. From the perspective of one of the hosts involved, a session is effectively identified by the 4-tuple (local IP, local port, remote IP, remote port). The *direction* of a session is normally the flow direction of the packet that initiates the session: the initial SYN packet for TCP, or the first user datagram for UDP.

Of the various flavors of NAT, the most common type is *traditional* or *outbound* NAT, which provides an asymmetric bridge between a private network and a public network. Outbound NAT by default allows only outbound sessions to traverse the NAT: incoming packets are dropped unless the NAT identifies them as being part of an existing session initiated from within the private network. Outbound NAT conflicts with peer-to-peer protocols because when both peers desiring to communicate are “behind” (on the private network side of) two different NATs, whichever peer tries to initiate a session, the other peer’s NAT rejects it. NAT traversal entails making P2P sessions look like “outbound” sessions to *both* NATs.

Outbound NAT has two sub-varieties: *Basic NAT*, which only translates IP addresses, and *Network Address/Port Translation* (NAPT), which translates entire session endpoints. NAPT, the more general variety, has also become the most common because it enables the hosts on a private network to share the use of a *single* public IP address. Throughout this paper we assume NAPT, though the principles and techniques we discuss apply equally well (if sometimes trivially) to Basic NAT.

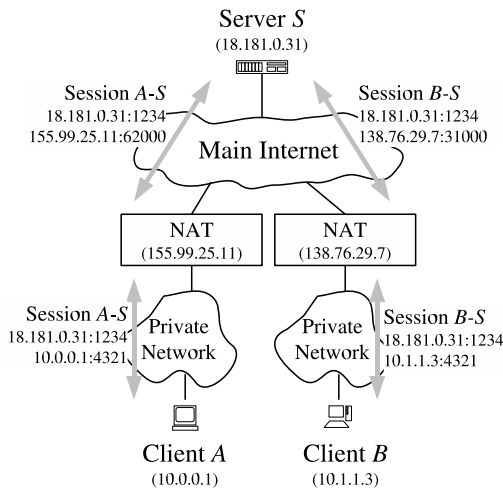


Figure 2: NAT Traversal by Relaying

2.2 Relaying

The most reliable—but least efficient—method of P2P communication across NAT is simply to make the communication look to the network like standard client/server communication, through relaying. Suppose two client hosts *A* and *B* have each initiated TCP or UDP connections to a well-known server *S*, at *S*’s global IP address 18.181.0.31 and port number 1234. As shown in Figure 2, the clients reside on separate private networks, and their respective NATs prevent either client from directly initiating a connection to the other. Instead of attempting a direct connection, the two clients can simply use the server *S* to relay messages between them. For example, to send a message to client *B*, client *A* simply sends the message to server *S* along its already-established client/server connection, and server *S* forwards the message on to client *B* using its existing client/server connection with *B*.

Relaying always works as long as both clients can connect to the server. Its disadvantages are that it consumes the server’s processing power and network bandwidth, and communication latency between the peering clients is likely increased even if the server is well-connected. Nevertheless, since there is no more efficient technique that works reliably on all existing NATs, relaying is a useful fall-back strategy if maximum robustness is desired. The TURN protocol [18] defines a method of implementing relaying in a relatively secure fashion.

2.3 Connection Reversal

Some P2P applications use a straightforward but limited technique, known as *connection reversal*, to enable communication when both hosts have connections to a well-

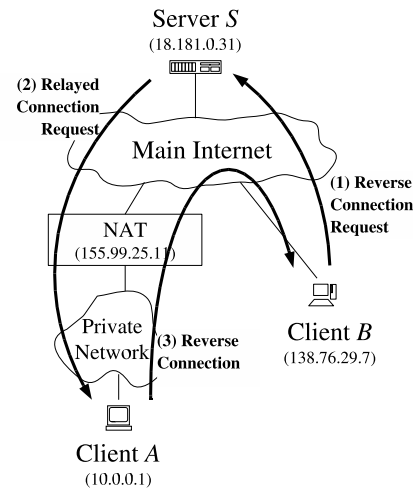


Figure 3: NAT Traversal by Connection Reversal

known rendezvous server *S* and only one of the peers is behind a NAT, as shown in Figure 3. If *A* wants to initiate a connection to *B*, then a direct connection attempt works automatically, because *B* is not behind a NAT and *A*’s NAT interprets the connection as an outgoing session. If *B* wants to initiate a connection to *A*, however, any direct connection attempt to *A* is blocked by *A*’s NAT. *B* can instead relay a connection request to *A* through a well-known server *S*, asking *A* to attempt a “reverse” connection back to *B*. Despite the obvious limitations of this technique, the central idea of using a well-known rendezvous server as an intermediary to help set up direct peer-to-peer connections is fundamental to the more general hole punching techniques described next.

3 UDP Hole Punching

UDP hole punching enables two clients to set up a direct peer-to-peer UDP session with the help of a well-known rendezvous server, even if the clients are both behind NATs. This technique was mentioned in section 5.1 of RFC 3027 [10], documented more thoroughly elsewhere on the Web [13], and used in recent experimental Internet protocols [17, 11]. Various proprietary protocols, such as those for on-line gaming, also use UDP hole punching.

3.1 The Rendezvous Server

Hole punching assumes that the two clients, *A* and *B*, already have active UDP sessions with a rendezvous server *S*. When a client registers with *S*, the server records *two* endpoints for that client: the (IP address, UDP port) pair that the client *believes* itself to be using to talk with *S*, and the (IP address, UDP port) pair that the server *ob-*

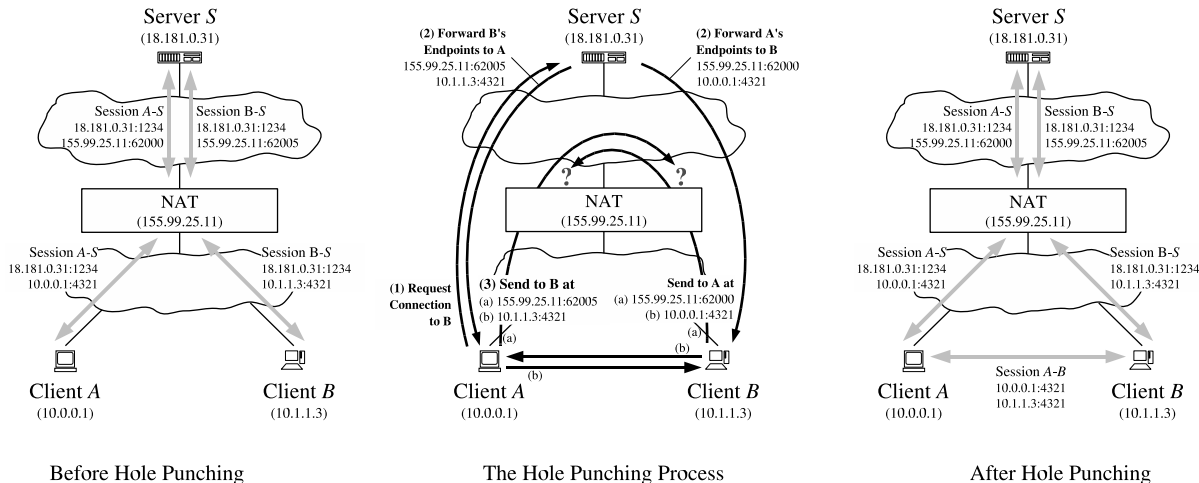


Figure 4: UDP Hole Punching, Peers Behind a Common NAT

serves the client to be using to talk with it. We refer to the first pair as the client’s *private* endpoint and the second as the client’s *public* endpoint. The server might obtain the client’s private endpoint from the client itself in a field in the body of the client’s registration message, and obtain the client’s public endpoint from the source IP address and source UDP port fields in the IP and UDP headers of that registration message. If the client is *not* behind a NAT, then its private and public endpoints should be identical.

A few poorly behaved NATs are known to scan the body of UDP datagrams for 4-byte fields that look like IP addresses, and translate them as they would the IP address fields in the IP header. To be robust against such behavior, applications may wish to obfuscate IP addresses in messages bodies slightly, for example by transmitting the one’s complement of the IP address instead of the IP address itself. Of course, if the application is encrypting its messages, then this behavior is not likely to be a problem.

3.2 Establishing Peer-to-Peer Sessions

Suppose client *A* wants to establish a UDP session directly with client *B*. Hole punching proceeds as follows:

1. *A* initially does not know how to reach *B*, so *A* asks *S* for help establishing a UDP session with *B*.
2. *S* replies to *A* with a message containing *B*’s public and private endpoints. At the same time, *S* uses its UDP session with *B* to send *B* a connection request message containing *A*’s public and private endpoints. Once these messages are received, *A* and *B* know each other’s public and private endpoints.
3. When *A* receives *B*’s public and private endpoints

from *S*, *A* starts sending UDP packets to *both* of these endpoints, and subsequently “locks in” whichever endpoint first elicits a valid response from *B*. Similarly, when *B* receives *A*’s public and private endpoints in the forwarded connection request, *B* starts sending UDP packets to *A* at each of *A*’s known endpoints, locking in the first endpoint that works. The order and timing of these messages are not critical as long as they are asynchronous.

We now consider how UDP hole punching handles each of three specific network scenarios. In the first situation, representing the “easy” case, the two clients actually reside behind the same NAT, on one private network. In the second, most common case, the clients reside behind different NATs. In the third scenario, the clients each reside behind *two* levels of NAT: a common “first-level” NAT deployed by an ISP for example, and distinct “second-level” NATs such as consumer NAT routers for home networks.

It is in general difficult or impossible for the application itself to determine the exact physical layout of the network, and thus which of these scenarios (or the many other possible ones) actually applies at a given time. Protocols such as STUN [19] can provide some information about the NATs present on a communication path, but this information may not always be complete or reliable, especially when multiple levels of NAT are involved. Nevertheless, hole punching works automatically in all of these scenarios *without* the application having to know the specific network organization, as long as the NATs involved behave in a reasonable fashion. (“Reasonable” behavior for NATs will be described later in Section 5.)

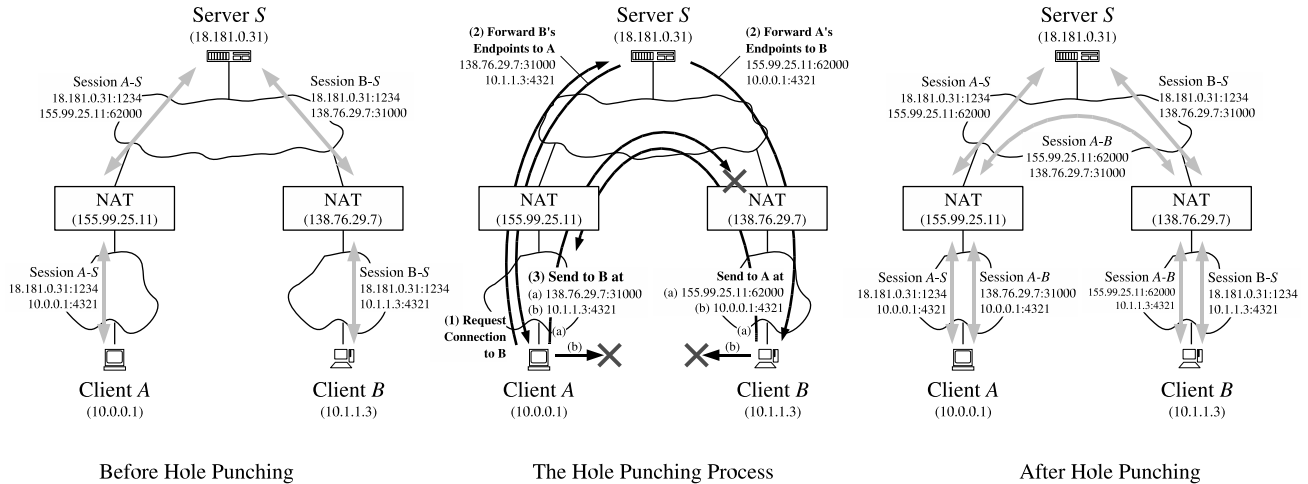


Figure 5: UDP Hole Punching, Peers Behind Different NATs

3.3 Peers Behind a Common NAT

First consider the simple scenario in which the two clients (probably unknowingly) happen to reside behind the same NAT, and are therefore located in the same private IP address realm, as shown in Figure 4. Client *A* has established a UDP session with server *S*, to which the common NAT has assigned its own public port number 62000. Client *B* has similarly established a session with *S*, to which the NAT has assigned public port number 62005.

Suppose that client *A* uses the hole punching technique outlined above to establish a UDP session with *B*, using server *S* as an introducer. Client *A* sends *S* a message requesting a connection to *B*. *S* responds to *A* with *B*'s public and private endpoints, and also forwards *A*'s public and private endpoints to *B*. Both clients then attempt to send UDP datagrams to each other directly at each of these endpoints. The messages directed to the public endpoints may or may not reach their destination, depending on whether or not the NAT supports hairpin translation as described below in Section 3.5. The messages directed at the private endpoints *do* reach their destinations, however, and since this direct route through the private network is likely to be faster than an indirect route through the NAT anyway, the clients are most likely to select the private endpoints for subsequent regular communication.

By assuming that NATs support hairpin translation, the application might dispense with the complexity of trying private as well as public endpoints, at the cost of making local communication behind a common NAT unnecessarily pass through the NAT. As our results in Section 6 show, however, hairpin translation is still much less common among existing NATs than are other “P2P-friendly” NAT

behaviors. For now, therefore, applications may benefit substantially by using both public and private endpoints.

3.4 Peers Behind Different NATs

Suppose clients *A* and *B* have private IP addresses behind different NATs, as shown in Figure 5. *A* and *B* have each initiated UDP communication sessions from their local port 4321 to port 1234 on server *S*. In handling these outbound sessions, NAT *A* has assigned port 62000 at its own public IP address, 155.99.25.11, for the use of *A*'s session with *S*, and NAT *B* has assigned port 31000 at its IP address, 138.76.29.7, to *B*'s session with *S*.

In *A*'s registration message to *S*, *A* reports its private endpoint to *S* as 10.0.0.1:4321, where 10.0.0.1 is *A*'s IP address on its own private network. *S* records *A*'s reported private endpoint, along with *A*'s public endpoint as observed by *S* itself. *A*'s public endpoint in this case is 155.99.25.11:62000, the temporary endpoint assigned to the session by the NAT. Similarly, when client *B* registers, *S* records *B*'s private endpoint as 10.1.1.3:4321 and *B*'s public endpoint as 138.76.29.7:31000.

Now client *A* follows the hole punching procedure described above to establish a UDP communication session directly with *B*. First, *A* sends a request message to *S* asking for help connecting with *B*. In response, *S* sends *B*'s public and private endpoints to *A*, and sends *A*'s public and private endpoints to *B*. *A* and *B* each start trying to send UDP datagrams directly to each of these endpoints.

Since *A* and *B* are on different private networks and their respective private IP addresses are not globally routable, the messages sent to these endpoints will reach either the wrong host or no host at all. Because many

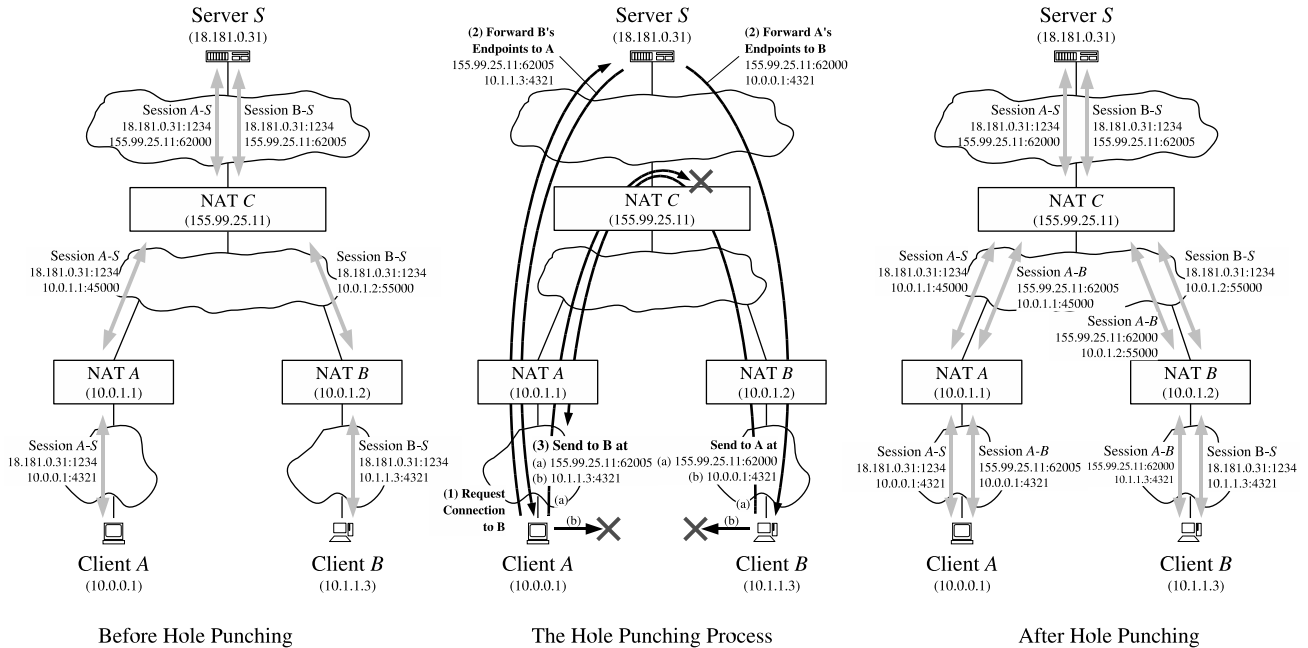


Figure 6: UDP Hole Punching, Peers Behind Multiple Levels of NAT

NATs also act as DHCP servers, handing out IP addresses in a fairly deterministic way from a private address pool usually determined by the NAT vendor by default, it is quite likely in practice that *A*'s messages directed at *B*'s private endpoint will reach *some* (incorrect) host on *A*'s private network that happens to have the same private IP address as *B* does. Applications must therefore authenticate all messages in some way to filter out such stray traffic robustly. The messages might include application-specific names or cryptographic tokens, for example, or at least a random nonce pre-arranged through *S*.

Now consider *A*'s first message sent to *B*'s public endpoint, as shown in Figure 5. As this outbound message passes through *A*'s NAT, this NAT notices that this is the first UDP packet in a new outgoing session. The new session's source endpoint (10.0.0.1:4321) is the same as that of the existing session between *A* and *S*, but its destination endpoint is different. If NAT *A* is well-behaved, it preserves the identity of *A*'s private endpoint, consistently translating *all* outbound sessions from private source endpoint 10.0.0.1:4321 to the corresponding public source endpoint 155.99.25.11:62000. *A*'s first outgoing message to *B*'s public endpoint thus, in effect, "punches a hole" in *A*'s NAT for a new UDP session identified by the endpoints (10.0.0.1:4321, 138.76.29.7:31000) on *A*'s private network, and by the endpoints (155.99.25.11:62000, 138.76.29.7:31000) on the main Internet.

If *A*'s message to *B*'s public endpoint reaches *B*'s NAT before *B*'s first message to *A* has crossed *B*'s own NAT, then *B*'s NAT may interpret *A*'s inbound message as unsolicited incoming traffic and drop it. *B*'s first message to *A*'s public address, however, similarly opens a hole in *B*'s NAT, for a new UDP session identified by the endpoints (10.1.1.3:4321, 155.99.25.11:62000) on *B*'s private network, and by the endpoints (138.76.29.7:31000, 155.99.25.11:62000) on the Internet. Once the first messages from *A* and *B* have crossed their respective NATs, holes are open in each direction and UDP communication can proceed normally. Once the clients have verified that the public endpoints work, they can stop sending messages to the alternative private endpoints.

3.5 Peers Behind Multiple Levels of NAT

In some topologies involving multiple NAT devices, two clients cannot establish an "optimal" P2P route between them without specific knowledge of the topology. Consider a final scenario, depicted in Figure 6. Suppose NAT *C* is a large industrial NAT deployed by an internet service provider (ISP) to multiplex many customers onto a few public IP addresses, and NATs *A* and *B* are small consumer NAT routers deployed independently by two of the ISP's customers to multiplex their private home networks onto their respective ISP-provided IP addresses. Only server *S* and NAT *C* have globally routable IP ad-

dresses; the “public” IP addresses used by NAT *A* and NAT *B* are actually private to the ISP’s address realm, while client *A*’s and *B*’s addresses in turn are private to the addressing realms of NAT *A* and NAT *B*, respectively. Each client initiates an outgoing connection to server *S* as before, causing NATs *A* and *B* each to create a single public/private translation, and causing NAT *C* to establish a public/private translation for each session.

Now suppose *A* and *B* attempt to establish a direct peer-to-peer UDP connection via hole punching. The optimal routing strategy would be for client *A* to send messages to client *B*’s “semi-public” endpoint at NAT *B*, 10.0.1.2:55000 in the ISP’s addressing realm, and for client *B* to send messages to *A*’s “semi-public” endpoint at NAT *B*, namely 10.0.1.1:45000. Unfortunately, *A* and *B* have no way to learn these addresses, because server *S* only sees the truly global public endpoints of the clients, 155.99.25.11:62000 and 155.99.25.11:62005 respectively. Even if *A* and *B* had some way to learn these addresses, there is still no guarantee that they would be usable, because the address assignments in the ISP’s private address realm might conflict with unrelated address assignments in the clients’ private realms. (NAT *A*’s IP address in NAT *C*’s realm might just as easily have been 10.1.1.3, for example, the same as client *B*’s private address in NAT *B*’s realm.)

The clients therefore have no choice but to use their global public addresses as seen by *S* for their P2P communication, and rely on NAT *C* providing *hairpin* or *loop-back* translation. When *A* sends a UDP datagram to *B*’s global endpoint, 155.99.25.11:62005, NAT *A* first translates the datagram’s source endpoint from 10.0.0.1:4321 to 10.0.1.1:45000. The datagram now reaches NAT *C*, which recognizes that the datagram’s destination address is one of NAT *C*’s own translated *public* endpoints. If NAT *C* is well-behaved, it then translates *both* the source and destination addresses in the datagram and “loops” the datagram back onto the private network, now with a source endpoint of 155.99.25.11:62000 and a destination endpoint of 10.0.1.2:55000. NAT *B* finally translates the datagram’s destination address as the datagram enters *B*’s private network, and the datagram reaches *B*. The path back to *A* works similarly. Many NATs do not yet support hairpin translation, but it is becoming more common as NAT vendors become aware of this issue.

3.6 UDP Idle Timeouts

Since the UDP transport protocol provides NATs with no reliable, application-independent way to determine the lifetime of a session crossing the NAT, most NATs simply associate an idle timer with UDP translations, closing the hole if no traffic has used it for some time period. There

is unfortunately no standard value for this timer: some NATs have timeouts as short as 20 seconds. If the application needs to keep an idle UDP session active after establishing the session via hole punching, the application must send periodic keep-alive packets to ensure that the relevant translation state in the NATs does not disappear.

Unfortunately, many NATs associate UDP idle timers with individual UDP sessions defined by a particular pair of endpoints, so sending keep-alives on one session will not keep other sessions active even if all the sessions originate from the same private endpoint. Instead of sending keep-alives on many different P2P sessions, applications can avoid excessive keep-alive traffic by detecting when a UDP session no longer works, and re-running the original hole punching procedure again “on demand.”

4 TCP Hole Punching

Establishing peer-to-peer TCP connections between hosts behind NATs is slightly more complex than for UDP, but TCP hole punching is remarkably similar at the protocol level. Since it is not as well-understood, it is currently supported by fewer existing NATs. When the NATs involved *do* support it, however, TCP hole punching is just as fast and reliable as UDP hole punching. Peer-to-peer TCP communication across well-behaved NATs may in fact be *more* robust than UDP communication, because unlike UDP, the TCP protocol’s state machine gives NATs on the path a standard way to determine the precise lifetime of a particular TCP session.

4.1 Sockets and TCP Port Reuse

The main practical challenge to applications wishing to implement TCP hole punching is not a protocol issue but an application programming interface (API) issue. Because the standard Berkeley sockets API was designed around the client/server paradigm, the API allows a TCP stream socket to be used to initiate an outgoing connection via `connect()`, or to listen for incoming connections via `listen()` and `accept()`, *but not both*. Further, TCP sockets usually have a one-to-one correspondence to TCP port numbers on the local host: after the application binds one socket to a particular local TCP port, attempts to bind a second socket to the same TCP port fail.

For TCP hole punching to work, however, we need to use a single local TCP port to listen for incoming TCP connections and to initiate multiple outgoing TCP connections concurrently. Fortunately, all major operating systems support a special TCP socket option, commonly named `SO_REUSEADDR`, which allows the application to bind multiple sockets to the same local endpoint as long as this option is set on all of the sockets involved. BSD

systems have introduced a `SO_REUSEPORT` option that controls port reuse separately from address reuse; on such systems *both* of these options must be set.

4.2 Opening Peer-to-Peer TCP Streams

Suppose that client *A* wishes to set up a TCP connection with client *B*. We assume as usual that both *A* and *B* already have active TCP connections with a well-known rendezvous server *S*. The server records each registered client's public and private endpoints, just as for UDP. At the protocol level, TCP hole punching works almost exactly as for UDP:

1. Client *A* uses its active TCP session with *S* to ask *S* for help connecting to *B*.
2. *S* replies to *A* with *B*'s public and private TCP endpoints, and at the same time sends *A*'s public and private endpoints to *B*.
3. From the *same local TCP ports* that *A* and *B* used to register with *S*, *A* and *B* each asynchronously make outgoing connection attempts to the other's public and private endpoints as reported by *S*, while simultaneously listening for incoming connections on their respective local TCP ports.
4. *A* and *B* wait for outgoing connection attempts to succeed, and/or for incoming connections to appear. If one of the outgoing connection attempts fails due to a network error such as "connection reset" or "host unreachable," the host simply re-tries that connection attempt after a short delay (e.g., one second), up to an application-defined maximum timeout period.
5. When a TCP connection is made, the hosts authenticate each other to verify that they connected to the intended host. If authentication fails, the clients close that connection and continue waiting for others to succeed. The clients use the first successfully authenticated TCP stream resulting from this process.

Unlike with UDP, where each client only needs one socket to communicate with both *S* and any number of peers simultaneously, with TCP each client application must manage several sockets bound to a single local TCP port on that client node, as shown in Figure 7. Each client needs a stream socket representing its connection to *S*, a listen socket on which to accept incoming connections from peers, and at least two additional stream sockets with which to initiate outgoing connections to the other peer's public and private TCP endpoints.

Consider the common-case scenario in which the clients *A* and *B* are behind different NATs, as shown in

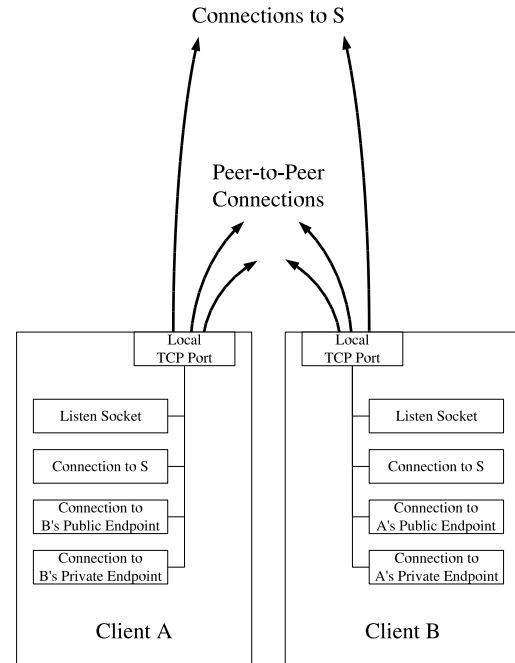


Figure 7: Sockets versus Ports for TCP Hole Punching

Figure 5, and assume that the port numbers shown in the figure are now for TCP rather than UDP ports. The outgoing connection attempts *A* and *B* make to each other's private endpoints either fail or connect to the wrong host. As with UDP, it is important that TCP applications authenticate their peer-to-peer sessions, due of the likelihood of mistakenly connecting to a random host on the local network that happens to have the same private IP address as the desired host on a remote private network.

The clients' outgoing connection attempts to each other's *public* endpoints, however, cause the respective NATs to open up new "holes" enabling direct TCP communication between *A* and *B*. If the NATs are well-behaved, then a new peer-to-peer TCP stream automatically forms between them. If *A*'s first SYN packet to *B* reaches *B*'s NAT before *B*'s first SYN packet to *A* reaches *B*'s NAT, for example, then *B*'s NAT may interpret *A*'s SYN as an unsolicited incoming connection attempt and drop it. *B*'s first SYN packet to *A* should subsequently get through, however, because *A*'s NAT sees this SYN as being part of the outbound session to *B* that *A*'s first SYN had already initiated.

4.3 Behavior Observed by the Application

What the client applications observe to happen with their sockets during TCP hole punching depends on the timing and the TCP implementations involved. Suppose that

A's first outbound SYN packet to *B*'s public endpoint is dropped by NAT *B*, but *B*'s first subsequent SYN packet to *A*'s public endpoint gets through to *A* before *A*'s TCP retransmits its SYN. Depending on the operating system involved, one of two things may happen:

- *A*'s TCP implementation notices that the session endpoints for the incoming SYN match those of an outbound session *A* was attempting to initiate. *A*'s TCP stack therefore associates this new session with the socket that the local application on *A* was using to `connect()` to *B*'s public endpoint. The application's asynchronous `connect()` call succeeds, and nothing happens with the application's listen socket.

Since the received SYN packet did not include an ACK for *A*'s previous outbound SYN, *A*'s TCP replies to *B*'s public endpoint with a SYN-ACK packet, the SYN part being merely a replay of *A*'s original outbound SYN, using the same sequence number. Once *B*'s TCP receives *A*'s SYN-ACK, it responds with its own ACK for *A*'s SYN, and the TCP session enters the connected state on both ends.

- Alternatively, *A*'s TCP implementation might instead notice that *A* has an active listen socket on that port waiting for incoming connection attempts. Since *B*'s SYN looks like an incoming connection attempt, *A*'s TCP creates a *new* stream socket with which to associate the new TCP session, and hands this new socket to the application via the application's next `accept()` call on its listen socket. *A*'s TCP then responds to *B* with a SYN-ACK as above, and TCP connection setup proceeds as usual for client/server-style connections.

Since *A*'s prior outbound `connect()` attempt to *B* used a combination of source and destination endpoints that is now in use by another socket, namely the one just returned to the application via `accept()`, *A*'s asynchronous `connect()` attempt must fail at some point, typically with an "address in use" error. The application nevertheless has the working peer-to-peer stream socket it needs to communicate with *B*, so it ignores this failure.

The first behavior above appears to be usual for BSD-based operating systems, whereas the second behavior appears more common under Linux and Windows.

4.4 Simultaneous TCP Open

Suppose that the timing of the various connection attempts during the hole punching process works out so that

the initial outgoing SYN packets from *both* clients traverse their respective local NATs, opening new outbound TCP sessions in each NAT, before reaching the remote NAT. In this "lucky" case, the NATs do not reject either of the initial SYN packets, and the SYNs cross on the wire between the two NATs. In this case, the clients observe an event known as a *simultaneous TCP open*: each peer's TCP receives a "raw" SYN while waiting for a SYN-ACK. Each peer's TCP responds with a SYN-ACK, whose SYN part essentially "replays" the peer's previous outgoing SYN, and whose ACK part acknowledges the SYN received from the other peer.

What the respective applications observe in this case again depends on the behavior of the TCP implementations involved, as described in the previous section. If *both* clients implement the second behavior above, it may be that *all* of the asynchronous `connect()` calls made by the application ultimately fail, but the application running on each client nevertheless receives a new, working peer-to-peer TCP stream socket via `accept()`—as if this TCP stream had magically "created itself" on the wire and was merely passively accepted at the endpoints! As long as the application does not care whether it ultimately receives its peer-to-peer TCP sockets via `connect()` or `accept()`, the process results in a working stream on any TCP implementation that properly implements the standard TCP state machine specified in RFC 793 [23].

Each of the alternative network organization scenarios discussed in Section 3 for UDP works in exactly the same way for TCP. For example, TCP hole punching works in multi-level NAT scenarios such as the one in Figure 6 as long as the NATs involved are well-behaved.

4.5 Sequential Hole Punching

In a variant of the above TCP hole punching procedure implemented by the NatTrav library [4], the clients attempt connections to each other sequentially rather than in parallel. For example: (1) *A* informs *B* via *S* of its desire to communicate, *without* simultaneously listening on its local port; (2) *B* makes a `connect()` attempt to *A*, which opens a hole in *B*'s NAT but then fails due to a timeout or RST from *A*'s NAT or a RST from *A* itself; (3) *B* closes its connection to *S* and does a `listen()` on its local port; (4) *S* in turn closes its connection with *A*, signaling *A* to attempt a `connect()` directly to *B*.

This sequential procedure may be particularly useful on Windows hosts prior to XP Service Pack 2, which did not correctly implement simultaneous TCP open, or on sockets APIs that do not support the `SO_REUSEADDR` functionality. The sequential procedure is more timing-dependent, however, and may be slower in the common case and less robust in unusual situations. In step (2), for

example, *B* must allow its “doomed-to-fail” `connect()` attempt enough time to ensure that at least one SYN packet traverses all NATs on its side of the network. Too little delay risks a lost SYN derailing the process, whereas too much delay increases the total time required for hole punching. The sequential hole punching procedure also effectively “consumes” both clients’ connections to the server *S*, requiring the clients to open fresh connections to *S* for each new P2P connection to be forged. The parallel hole punching procedure, in contrast, typically completes as soon as both clients make their outgoing `connect()` attempts, and allows each client to retain and re-use a single connection to *S* indefinitely.

5 Properties of P2P-Friendly NATs

This section describes the key behavioral properties NATs must have in order for the hole punching techniques described above to work properly. Not all current NAT implementations satisfy these properties, but many do, and NATs are gradually becoming more “P2P-friendly” as NAT vendors recognize the demand for peer-to-peer protocols such as voice over IP and on-line gaming.

This section is not meant to be a complete or definitive specification for how NATs “should” behave; we provide it merely for information about the most commonly observed behaviors that enable or break P2P hole punching. The IETF has started a new working group, BEHAVE, to define official “best current practices” for NAT behavior. The BEHAVE group’s initial drafts include the considerations outlined in this section and others; NAT vendors should of course follow the IETF working group directly as official behavioral standards are formulated.

5.1 Consistent Endpoint Translation

The hole punching techniques described here only work automatically if the NAT consistently maps a given TCP or UDP source endpoint on the private network to a *single* corresponding public endpoint controlled by the NAT. A NAT that behaves in this way is referred to as a *cone NAT* in RFC 3489 [19] and elsewhere, because the NAT “focuses” all sessions originating from a single private endpoint through the same public endpoint on the NAT.

Consider again the scenario in Figure 5, for example. When client *A* initially contacted the well-known server *S*, NAT *A* chose to use port 62000 at its own public IP address, 155.99.25.11, as a temporary public endpoint to representing *A*’s private endpoint 10.0.0.1:4321. When *A* later attempts to establish a peer-to-peer session with *B* by sending a message from the same local private endpoint to *B*’s public endpoint, *A* depends on NAT *A* preserving the identity of this private endpoint, and re-using the exist-

ing public endpoint of 155.99.25.11:62000, because that is the public endpoint for *A* to which *B* will be sending its corresponding messages.

A NAT that is only designed to support client/server protocols will not necessarily preserve the identities of private endpoints in this way. Such a NAT is a *symmetric NAT* in RFC 3489 terminology. For example, after the NAT assigns the public endpoint 155.99.25.11:62000 to client *A*’s session with server *S*, the NAT might assign a different public endpoint, such as 155.99.25.11:62001, to the P2P session that *A* tries to initiate with *B*. In this case, the hole punching process fails to provide connectivity, because the subsequent incoming messages from *B* reach NAT *A* at the wrong port number.

Many symmetric NATs allocate port numbers for successive sessions in a fairly predictable way. Exploiting this fact, variants of hole punching algorithms [9, 1] can be made to work “much of the time” even over symmetric NATs by first probing the NAT’s behavior using a protocol such as STUN [19], and using the resulting information to “predict” the public port number the NAT will assign to a new session. Such prediction techniques amount to chasing a moving target, however, and many things can go wrong along the way. The predicted port number might already be in use causing the NAT to jump to another port number, for example, or another client behind the same NAT might initiate an unrelated session at the wrong time so as to allocate the predicted port number. While port number prediction can be a useful trick for achieving maximum compatibility with badly-behaved existing NATs, it does not represent a robust long-term solution. Since symmetric NAT provides no greater security than a cone NAT with per-session traffic filtering, symmetric NAT is becoming less common as NAT vendors adapt their algorithms to support P2P protocols.

5.2 Handling Unsolicited TCP Connections

When a NAT receives a SYN packet on its public side for what appears to be an unsolicited incoming connection attempt, it is important that the NAT just silently drop the SYN packet. Some NATs instead actively reject such incoming connections by sending back a TCP RST packet or even an ICMP error report, which interferes with the TCP hole punching process. Such behavior is not necessarily fatal, as long as the applications re-try outgoing connection attempts as specified in step 4 of the process described in Section 4.2, but the resulting transient errors can make hole punching take longer.

5.3 Leaving Payloads Alone

A few existing NATs are known to scan “blindly” through packet payloads for 4-byte values that look like IP ad-

addresses, and translate them as they would the IP address in the packet header, without knowing anything about the application protocol in use. This bad behavior fortunately appears to be uncommon, and applications can easily protect themselves against it by obfuscating IP addresses they send in messages, for example by sending the bitwise complement of the desired IP address.

5.4 Hairpin Translation

Some multi-level NAT situations require hairpin translation support in order for either TCP or UDP hole punching to work, as described in Section 3.5. The scenario shown in Figure 6, for example, depends on NAT *C* providing hairpin translation. Support for hairpin translation is unfortunately rare in current NATs, but fortunately so are the network scenarios that require it. Multi-level NAT is becoming more common as IPv4 address space depletion continues, however, so support for hairpin translation is important in future NAT implementations.

6 Evaluation of Existing NATs

To evaluate the robustness of the TCP and UDP hole punching techniques described in this paper on a variety of existing NATs, we implemented and distributed a test program called NAT Check [16], and solicited data from Internet users about their NATs.

NAT Check's primary purpose is to test NATs for the two behavioral properties most crucial to reliable UDP and TCP hole punching: namely, consistent identity-preserving endpoint translation (Section 5.1), and silently dropping unsolicited incoming TCP SYNs instead of rejecting them with RSTs or ICMP errors (Section 5.2). In addition, NAT Check separately tests whether the NAT supports hairpin translation (Section 5.4), and whether the NAT filters unsolicited incoming traffic at all. This last property does not affect hole punching, but provides a useful indication the NAT's firewall policy.

NAT Check makes no attempt to test every relevant facet of NAT behavior individually: a wide variety of subtle behavioral differences are known, some of which are difficult to test reliably [12]. Instead, NAT Check merely attempts to answer the question, "how commonly can the proposed hole punching techniques be expected to work on deployed NATs, under typical network conditions?"

6.1 Test Method

NAT Check consists of a client program to be run on a machine behind the NAT to be tested, and three well-known servers at different global IP addresses. The client cooperates with the three servers to check the NAT behavior relevant to both TCP and UDP hole punching. The client

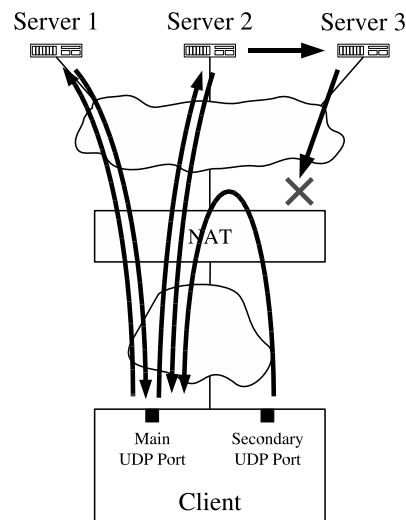


Figure 8: NAT Check Test Method for UDP

program is small and relatively portable, currently running on Windows, Linux, BSD, and Mac OS X. The machines hosting the well-known servers all run FreeBSD.

6.1.1 UDP Test

To test the NAT's behavior for UDP, the client opens a socket and binds it to a local UDP port, then successively sends "ping"-like requests to servers 1 and 2, as shown in Figure 8. These servers each respond to the client's pings with a reply that includes the client's public UDP endpoint: the client's own IP address and UDP port number as observed by the server. If the two servers report the same public endpoint for the client, NAT Check assumes that the NAT properly preserves the identity of the client's private endpoint, satisfying the primary precondition for reliable UDP hole punching.

When server 2 receives a UDP request from the client, besides replying directly to the client it also forwards the request to server 3, which in turn replies to the client from its own IP address. If the NAT's firewall properly filters "unsolicited" incoming traffic on a per-session basis, then the client never sees these replies from server 3, even though they are directed at the same public port as the replies from servers 1 and 2.

To test the NAT for hairpin translation support, the client simply opens a second UDP socket at a different local port and uses it to send messages to the *public* endpoint representing the client's first UDP socket, as reported by server 2. If these messages reach the client's first private endpoint, then the NAT supports hairpin translation.

6.1.2 TCP Test

The TCP test follows a similar pattern as for UDP. The client uses a single local TCP port to initiate outbound sessions to servers 1 and 2, and checks whether the public endpoints reported by servers 1 and 2 are the same, the first precondition for reliable TCP hole punching.

The NAT's response to unsolicited incoming connection attempts also impacts the speed and reliability of TCP hole punching, however, so NAT Check also tests this behavior. When server 2 receives the client's request, instead of immediately replying to the client, it forwards a request to server 3 and waits for server 3 to respond with a "go-ahead" signal. When server 3 receives this forwarded request, it attempts to initiate an inbound connection to the client's public TCP endpoint. Server 3 waits up to five seconds for this connection to succeed or fail, and if the connection attempt is still "in progress" after five seconds, server 3 responds to server 2 with the "go-ahead" signal and continues waiting for up to 20 seconds. Once the client finally receives server 2's reply (which server 2 delayed waiting for server 3's "go-ahead" signal), the client attempts an outbound connection to server 3, effectively causing a simultaneous TCP open with server 3.

What happens during this test depends on the NAT's behavior as follows. If the NAT properly just drops server 3's "unsolicited" incoming SYN packets, then nothing happens on the client's listen socket during the five second period before server 2 replies to the client. When the client finally initiates its own connection to server 3, opening a hole through the NAT, the attempt succeeds immediately. If on the other hand the NAT does *not* drop server 3's unsolicited incoming SYNs but allows them through (which is fine for hole punching but not ideal for security), then the client receives an incoming TCP connection on its listen socket before receiving server 2's reply. Finally, if the NAT actively rejects server 3's unsolicited incoming SYNs by sending back TCP RST packets, then server 3 gives up and the client's subsequent attempt to connect to server 3 fails.

To test hairpin translation for TCP, the client simply uses a secondary local TCP port to attempt a connection to the public endpoint corresponding to its primary TCP port, in the same way as for UDP.

6.2 Test Results

The NAT Check data we gathered consists of 380 reported data points covering a variety of NAT router hardware from 68 vendors, as well as the NAT functionality built into different versions of eight popular operating systems. Only 335 of the total data points include results for UDP hairpin translation, and only 286 data points include re-

sults for TCP, because we implemented these features in later versions of NAT Check after we had already started gathering results. The data is summarized by NAT vendor in Table 1; the table only individually lists vendors for which at least five data points were available. The variations in the test results for a given vendor can be accounted for by a variety of factors, such as different NAT devices or product lines sold by the same vendor, different software or firmware versions of the same NAT implementation, different configurations, and probably occasional NAT Check testing or reporting errors.

Out of the 380 reported data points for UDP, in 310 cases (82%) the NAT consistently translated the client's private endpoint, indicating basic compatibility with UDP hole punching. Support for hairpin translation is much less common, however: of the 335 data points that include UDP hairpin translation results, only 80 (24%) show hairpin translation support.

Out of the 286 data points for TCP, 184 (64%) show compatibility with TCP hole punching: the NAT consistently translates the client's private TCP endpoint, and does not send back RST packets in response to unsolicited incoming connection attempts. Hairpin translation support is again much less common: only 37 (13%) of the reports showed hairpin support for TCP.

Since these reports were generated by a "self-selecting" community of volunteers, they do not constitute a random sample and thus do not necessarily represent the true distribution of the NATs in common use. The results are nevertheless encouraging: it appears that the majority of commonly-deployed NATs already support UDP and TCP hole punching at least in single-level NAT scenarios.

6.3 Testing Limitations

There are a few limitations in NAT Check's current testing protocol that may cause misleading results in some cases. First, we only learned recently that a few NAT implementations blindly translate IP addresses they find in unknown application payloads, and the NAT Check protocol currently does not protect itself from this behavior by obfuscating the IP addresses it transmits.

Second, NAT Check's current hairpin translation checking may yield unnecessarily pessimistic results because it does not use the full, two-way hole punching procedure for this test. NAT Check currently assumes that a NAT supporting hairpin translation does not filter "incoming" hairpin connections arriving from the private network in the way it would filter incoming connections arriving at the public side of the NAT, because such filtering is unnecessary for security. We later realized, however, that a NAT might simplistically treat *any* traffic directed at the NAT's public ports as "untrusted" regardless of its origin. We do

	UDP				TCP			
	Hole Punching		Hairpin		Hole Punching		Hairpin	
NAT Hardware								
Linksys	45/46	(98%)	5/42	(12%)	33/38	(87%)	3/38	(8%)
Netgear	31/37	(84%)	3/35	(9%)	19/30	(63%)	0/30	(0%)
D-Link	16/21	(76%)	11/21	(52%)	9/19	(47%)	2/19	(11%)
Draytek	2/17	(12%)	3/12	(25%)	2/7	(29%)	0/7	(0%)
Belkin	14/14	(100%)	1/14	(7%)	11/11	(100%)	0/11	(0%)
Cisco	12/12	(100%)	3/9	(33%)	6/7	(86%)	2/7	(29%)
SMC	12/12	(100%)	3/10	(30%)	8/9	(89%)	2/9	(22%)
ZyXEL	7/9	(78%)	1/8	(13%)	0/7	(0%)	0/7	(0%)
3Com	7/7	(100%)	1/7	(14%)	5/6	(83%)	0/6	(0%)
OS-based NAT								
Windows	31/33	(94%)	11/32	(34%)	16/31	(52%)	28/31	(90%)
Linux	26/32	(81%)	3/25	(12%)	16/24	(67%)	2/24	(8%)
FreeBSD	7/9	(78%)	3/6	(50%)	2/3	(67%)	1/1	(100%)
All Vendors	310/380	(82%)	80/335	(24%)	184/286	(64%)	37/286	(13%)

Table 1: User Reports of NAT Support for UDP and TCP Hole Punching

not yet know which behavior is more common.

Finally, NAT implementations exist that consistently translate the client's private endpoint as long as *only one* client behind the NAT is using a particular private port number, but switch to symmetric NAT or even worse behaviors if two or more clients with different IP addresses on the private network try to communicate through the NAT from the same private port number. NAT Check could only detect this behavior by requiring the user to run it on two or more client hosts behind the NAT at the same time. Doing so would make NAT Check much more difficult to use, however, and impossible for users who only have one usable machine behind the NAT. Nevertheless, we plan to implement this testing functionality as an option in a future version of NAT Check.

6.4 Corroboration of Results

Despite testing difficulties such as those above, our results are generally corroborated by those of a large ISP, who recently found that of the top three consumer NAT router vendors, representing 86% of the NATs observed on their network, all three vendors currently produce NATs compatible with UDP hole punching [25]. Additional independent results recently obtained using the UDP-oriented STUN protocol [12], and STUNT, a TCP-enabled extension [8, 9], also appear consistent with our results. These latter studies provide more information on each NAT by testing a wider variety of behaviors individually, instead of just testing for basic hole punching compatibility as

NAT Check does. Since these more extensive tests require multiple cooperating clients behind the NAT and thus are more difficult to run, however, these results are so far available on a more limited variety of NATs.

7 Related Work

UDP hole punching was first explored and publicly documented by Dan Kegel [13], and is by now well-known in peer-to-peer application communities. Important aspects of UDP hole punching have also been indirectly documented in the specifications of several experimental protocols, such as STUN [19], ICE [17], and Teredo [11]. We know of no existing published work that thoroughly analyzes hole punching, however, or that points out the hairpin translation issue for multi-level NAT (Section 3.5).

We also know of no prior work that develops TCP hole punching in the symmetric fashion described here. Even the existence of the crucial `SO_REUSEADDR`/`SO_REUSEPORT` options in the Berkeley sockets API appears to be little-known among P2P application developers. NatTrav [4] implements a similar but asymmetric TCP hole punching procedure outlined earlier in Section 4.5. NUTSS [9] and NATBLASTER [1] implement more complex TCP hole punching tricks that can work around some of the bad NAT behaviors mentioned in Section 5, but they require the rendezvous server to spoof source IP addresses, and they also require the client applications to have access to “raw” sockets, usually available only at root or administrator privilege levels.

Protocols such as SOCKS [14], UPnP [26], and MIDCOM [22] allow applications to traverse a NAT through explicit cooperation with the NAT. These protocols are not widely or consistently supported by NAT vendors or applications, however, and do not appear to address the increasingly important multi-level NAT scenarios. Explicit control of a NAT further requires the application to locate the NAT and perhaps authenticate itself, which typically involves explicit user configuration. When hole punching works, in contrast, it works with no user intervention.

Recent proposals such as HIP [15] and FARA [2] extend the Internet's basic architecture by decoupling a host's identity from its location [20]. IPNL [7], UIP [5, 6], and DOA [27] propose schemes for routing across NATs in such an architecture. While such extensions are probably needed in the long term, hole punching enables applications to work over the existing network infrastructure immediately with no protocol stack upgrades, and leaves the notion of "host identity" for applications to define.

8 Conclusion

Hole punching is a general-purpose technique for establishing peer-to-peer connections in the presence of NAT. As long as the NATs involved meet certain behavioral requirements, hole punching works consistently and robustly for both TCP and UDP communication, and can be implemented by ordinary applications with no special privileges or specific network topology information. Hole punching fully preserves the transparency that is one of the most important hallmarks and attractions of NAT, and works even with multiple levels of NAT—though certain corner case situations require hairpin translation, a NAT feature not yet widely implemented.

Acknowledgments

The authors wish to thank Dave Andersen for his crucial support in gathering the results presented in Section 6. We also wish to thank Henrik Nordstrom, Christian Huitema, Justin Uberti, Mema Roussopoulos, and the anonymous USENIX reviewers for valuable feedback on early drafts of this paper. Finally, we wish to thank the many volunteers who took the time to run NAT Check on their systems and submit the results.

References

- [1] Andrew Biggadike, Daniel Ferullo, Geoffrey Wilson, and Adrian Perrig. NATBLASTER: Establishing TCP connections between hosts behind NATs. In *ACM SIGCOMM Asia Workshop*, Beijing, China, April 2005.
- [2] David Clark, Robert Braden, Aaron Falk, and Venkata Pingali. FARA: Reorganizing the addressing architecture. In *ACM SIGCOMM FDNA Workshop*, August 2003.
- [3] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification, December 1998. RFC 2460.
- [4] Jeffrey L. Eppinger. TCP connections for P2P apps: A software approach to solving the NAT problem. Technical Report CMU-ISRI-05-104, Carnegie Mellon University, January 2005.
- [5] Bryan Ford. Scalable Internet routing on topology-independent node identities. Technical Report MIT-LCS-TR-926, MIT Laboratory for Computer Science, October 2003.
- [6] Bryan Ford. Unmanaged internet protocol: Taming the edge network management crisis. In *Second Workshop on Hot Topics in Networks*, Cambridge, MA, November 2003.
- [7] Paul Francis and Ramakrishna Gummadi. IPNL: A NAT-extended Internet architecture. In *ACM SIGCOMM*, August 2002.
- [8] Saikat Guha and Paul Francis. Simple traversal of UDP through NATs and TCP too (STUNT). <http://nutss.gforge.cis.cornell.edu/>.
- [9] Saikat Guha, Yutaka Takeda, and Paul Francis. NUTSS: A SIP-based approach to UDP and TCP network connectivity. In *SIGCOMM 2004 Workshops*, August 2004.
- [10] M. Holdrege and P. Srisuresh. Protocol complications with the IP network address translator, January 2001. RFC 3027.
- [11] C. Huitema. Teredo: Tunneling IPv6 over UDP through NATs, March 2004. Internet-Draft (Work in Progress).
- [12] C. Jennings. NAT classification results using STUN, October 2004. Internet-Draft (Work in Progress).
- [13] Dan Kegel. NAT and peer-to-peer networking, July 1999. <http://www.alumni.caltech.edu/~dank/peer-nat.html>.
- [14] M. Leech et al. SOCKS protocol, March 1996. RFC 1928.
- [15] R. Moskowitz and P. Nikander. Host identity protocol architecture, April 2003. Internet-Draft (Work in Progress).
- [16] NAT check. <http://midcom-p2p.sourceforge.net/>.
- [17] J. Rosenberg. Interactive connectivity establishment (ICE), October 2003. Internet-Draft (Work in Progress).
- [18] J. Rosenberg, C. Huitema, and R. Mahy. Traversal using relay NAT (TURN), October 2003. Internet-Draft (Work in Progress).
- [19] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - simple traversal of user datagram protocol (UDP) through network address translators (NATs), March 2003. RFC 3489.
- [20] J. Saltzer. On the naming and binding of network destinations. In P. Ravasio et al., editor, *Local Computer Networks*, pages 311–317. North-Holland, Amsterdam, 1982. RFC 1498.
- [21] P. Srisuresh and M. Holdrege. IP network address translator (NAT) terminology and considerations, August 1999. RFC 2663.
- [22] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan. Middlebox communication architecture and framework, August 2002. RFC 3303.
- [23] Transmission control protocol, September 1981. RFC 793.
- [24] G. Tsirtsis and P. Srisuresh. Network address translation - protocol translation (NAT-PT), February 2000. RFC 2766.
- [25] Justin Uberti. E-mail on IETF MIDCOM mailing list, February 2004. Message-ID: <402CEB11.1060906@aol.com>.
- [26] UPnP Forum. Internet gateway device (IGD) standardized device control protocol, November 2001. <http://www.upnp.org/>.
- [27] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes no longer considered harmful. In *USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

Maintaining High Bandwidth under Dynamic Network Conditions

Dejan Kostić, Ryan Braud, Charles Killian, Erik Vandekieft, James W. Anderson,
Alex C. Snoeren and Amin Vahdat *

*Department of Computer Science and Engineering
University of California, San Diego*

{dkostic, rbraud, ckillian, evandeki, jwanderson, snoeren, vahdat}@cs.ucsd.edu

Abstract

The need to distribute large files across multiple wide-area sites is becoming increasingly common, for instance, in support of scientific computing, configuring distributed systems, distributing software updates such as open source ISOs or Windows patches, or disseminating multimedia content. Recently a number of techniques have been proposed for simultaneously retrieving portions of a file from multiple remote sites with the twin goals of filling the client's pipe and overcoming any performance bottlenecks between the client and any individual server. While there are a number of interesting tradeoffs in locating appropriate download sites in the face of dynamically changing network conditions, to date there has been no systematic evaluation of the merits of different protocols. This paper explores the design space of file distribution protocols and conducts a detailed performance evaluation of a number of competing systems running in both controlled emulation environments and live across the Internet. Based on our experience with these systems under a variety of conditions, we propose, implement and evaluate Bullet' (Bullet prime), a mesh based high bandwidth data dissemination system that outperforms previous techniques under both static and dynamic conditions.

1 Introduction

The rapid, reliable, and efficient transmission of a data object from a single source to a large number of receivers spread across the Internet has long been the subject of research and development spanning computer systems, networking, algorithms, and theory. Initial attempts at addressing this problem focused on IP multicast, a network level primitive for constructing efficient IP-level trees to deliver individual packets from the source to each receiver. Fundamental problems with reliability, congestion control, heterogeneity, and deployment limited the widespread success and availability of IP multicast.

Building on the experience gained from IP multicast, a number of efforts then focused on building *application-level overlays* [2, 9, 10, 12, 24] where a source would transmit the content, for instance over multiple TCP connections, to a set of receivers that would implicitly act as interior nodes in a tree built up at the application level.

While certainly advancing the state of the art, tree-based techniques face fundamental reliability and performance limitations. First, bandwidth down an overlay tree is guaranteed to be monotonically decreasing. For instance, a single packet loss in a TCP transmission high up in a tree can significantly impact the throughput to all participants rooted at the node experiencing the loss. Hence, in a large overlay, participants near the leaves are likely to experience more limited bandwidth. Further, because each node only receives data from its parent, the failure of a single node can dramatically impact overall system reliability and may cause scalability concerns under churn as a potentially large number of nodes attempt to rejoin the tree. Reliability in overlay trees is perhaps even more of a concern than in IP-level multicast, since end hosts are more likely to fail or to suffer from performance anomalies relative to IP routers in the interior of the network.

Thus, "third-generation" data dissemination infrastructures have most recently focused on building application-level *overlay meshes* [3, 5, 13, 25]. The idea here is that end hosts self-organize into a general mesh by selecting multiple application-level peers. Each peer transmits a disjoint set of the target underlying data to the node, enabling the potential to greater resiliency to failure and increased performance. Reliability is improved because the failure of any single peer will typically only reduce the transmitted bandwidth by $1/n$ where n is the total number of peers being maintained by a given node. In fact, with a sufficient numbers of peers, the failure of any one peer may go unnoticed because the remaining peers may be able to quickly make up for the lost bandwidth. This additional redundancy also enables each

*This research is supported in part by the Center for Networked Systems, Hewlett Packard, IBM, and Intel. Vahdat and Snoeren are supported by NSF CAREER awards (CCR-9984328, CNS-0347949).

node to regenerate its peer set in a more leisurely manner, eliminating the “reconnection storm” that can result from the failure of a node in an overlay tree. Mesh-based approaches may also improve overall bandwidth delivered to all nodes as data can flow to receivers over multiple disjoint paths. It is straightforward to construct scenarios where two peers can deliver higher aggregate bandwidth to a given node than any one can as a result of bottlenecks in the middle of the network.

While a number of these mesh-based systems have demonstrated performance and reliability improvements over previous tree-based techniques, there is currently no detailed understanding of the relative merits of existing systems nor an understanding of the set of design issues that most significantly affect end-to-end system performance and reliability. We set out to explore this space, beginning with our own implementation of Bullet [13]. Our detailed performance comparison of these systems both live across PlanetLab [20] and in the ModelNet [28] emulation environment led us to identify a number of key design questions that appear to most significantly impact overall system performance and reliability, including: i) whether data is pushed from the source to receivers or pulled by individual receivers based on discovering missing data items, ii) peer selection algorithms where nodes determine the number and identity of peers likely to have a good combination of high bandwidth and a large volume of missing data items, iii) the strategy for determining which data items to request and in what quantity based on changing network conditions, iv) the strategy employed by the sender to transmit blocks to its current set of peers based on the data items currently queued for transmission, and v) whether the data stream should be encoded using, for instance, Erasure codes [17] or transmitted unmodified.

By examining these questions in detail, we designed and implemented *Bullet'* (pronounced as Bullet prime), a new mesh-based data dissemination protocol that outperforms existing systems, including Bullet, SplitStream, and BitTorrent, by 25-70% in terms of download times, under a variety of network conditions. Thus, the principal contribution of this work is an exploration and initial enumeration of the design space for a broad range of mesh-based overlays and the design, implementation, and evaluation of a system that we believe embodies some of the “best practices” in this space, enabling good performance and reliability under a variety of network conditions, ranging from the static to the highly dynamic.

It is becoming increasingly important to synchronize updates to a logical data volume across the Internet, such as synchronizing FTP mirrors or updating installed software in distributed testbeds such as PlanetLab and the Grid. Today, the state of the art for disseminating updates consists of having all clients contact a central repository

known to contain the most current version of the data to retrieve any updates. Unfortunately, limited bandwidth and CPU at any central repository limit the speed and reliability with which the data can be synchronized across a large number of sites. We designed and implemented Shotgun, a set of extensions to the popular `rsync` tool to enable clients to synchronize their state with a centralized server orders of magnitude more quickly than previously possible.

The rest of this paper is organized as follows. Section 2 describes the design space for all systems doing large scale file distribution, Section 3 describes the specific implementation and architecture of Bullet', and Section 4 presents the results of our experiments testing our experimental strategies and comparisons with other systems, including evaluation of Shotgun. Section 5 outlines the related work in the field, and the paper concludes in Section 6.

2 Design Principles and Patterns

Throughout the paper, we assume that the source transmits the file as a sequence of *blocks*, that serve as the smallest transfer unit. Otherwise, peers would not be able to help each other until they had the entire file downloaded. In addition, we concentrate on the case when the source of the file is the only node that has the file initially, and wishes to disseminate it to a large group of receivers as quickly as possible. This usage scenario corresponds to a flash-crowd retrieving a popular file over the Internet. In our terminology for peering relationships, a *sender* is a node sending data to a *receiver*.

The design of any large-scale file distribution system can be realized as careful decisions made regarding the fundamental tenets of peer-to-peer applications and data transfer. As we see them, these tenets are push vs. pull, encoding data, finding the right peers, methods for requesting data from those peers, and serving data to others in an effective manner. Additionally, an important design consideration is whether the system should be fair to participants or focus on being fast. In all cases, however, the underlying goal of downloading files is *always keep your incoming pipe full of new data*. This means that nodes must prevent duplicate data transmission, restrict control overhead in favor of distributing data, and adapt to changing network conditions. In the following sections we enumerate these fundamental decisions which all systems disseminating objects must make in more detail.

2.1 Push or Pull

Options for distributing the file can be categorized by whether data is pushed from sources to destinations,

pulled from sources by destinations, or a hybrid of the two. Traditional streaming applications typically choose the push methodology because data has low latency, is coming at a constant rate, and all participants are supposed to get all the data at the same time. To increase capacity, nodes may have multiple senders to them, and in a push system, they must then devote overhead to keeping their senders informed about what they have to prevent receipt of duplicates. Alternately, systems can use a pull mechanism where receivers must first learn of what data exists and which nodes have it, and then request it. This has the advantage that receivers, not senders, are in control of the size and specification of which data is outstanding to them, which allows them to control and adapt to their environments more easily. However, this two-step process of discovery followed by requesting means additional delay before receiving data and extra control messages in some cases.

2.2 Encoded or Unencoded

The simplest way of sending the file involves just sending the original, or *unencoded*, blocks into the overlay. An advantage of this approach is that receivers typically do not have to fit the entire file into physical memory to sustain high-performance. Incoming blocks can be cached in memory, and later written to disk. Blocks only have to be read when the peers request them. Even if the blocks are not in memory and have to be fetched from the disk, pipelining techniques can be used to absorb the cost of disk reads. As a downside, sending the unencoded file might expose the “last block” problem of some file distribution mechanisms, when it becomes difficult for a node to locate and retrieve the last few blocks of the file.

Recently, a number of erasure-correcting codes that implement the “digital fountain” [4] approach were suggested by researchers [15, 16, 17]. When the source encodes the file with these codes, *any* $(1 + \epsilon)n$ correctly received *encoded* blocks are sufficient to reconstruct the original n blocks, with the typically low *reception overhead* (ϵ) of 0.03 – 0.05. These codes hold the potential of removing the “last block” problem, because there is no need for a receiver to acquire any particular block, as long as it recovers a sufficient number of distinct blocks. In this paper, we assume that only the source is capable of encoding the file, and do not consider the potential benefits of *network coding* [1], where intermediate nodes can produce encoded packets from the ones they have received thus far.

Based on the publicly available specification, we implemented *rateless* erasure codes [17]. Although it is straightforward to implement these codes with a low CPU encoding and decoding overhead, they exhibit some performance artifacts that are relevant from a systems

perspective. First, the reconstruction of the file cannot make significant progress until a significant number of the encoded blocks is successfully received. Even with n received blocks (i.e., corresponding to the original file size), only 30 percent of the file content can be reconstructed [14]. Further, since the encoded blocks are computed by XOR-ing random sets of original blocks, the decoding stage requires random access to all of the reconstructed file blocks. This pattern of access, coupled with the bursty nature of the decoding process, causes increased decoding time due to excessive disk swapping if all of the decoded file blocks cannot fit into physical memory¹. Consequently, the source is forced to transmit the file as a series of *segments* that can fit into physical memory of the receivers. Even if all receivers have homogeneous memory size, this arrangement presents few problems. First, the source needs to decide when to start transmitting encoded blocks that belong to the next segment. If the file distribution mechanism exhibits considerable latency between the time a block is first generated and the time when nodes receive it, the source might send too many unnecessary blocks into the overlay. Second, receivers need to simultaneously locate and retrieve data belonging to multiple segments. Opening too many TCP connections can also affect overall performance. Therefore, the receivers have to locate enough senders hosting the segments they are interested in, while still being able to fill their incoming pipe.

We have observed a 4 percent overhead when encoding and decoding files of tens of MBs in size. Although mathematically possible, it is difficult to make this overhead arbitrary small via parameter settings or a large number of blocks. Since the file blocks should be sufficiently large to overcome the fixed overhead due to per-block headers, we cannot use an excessive number of blocks. Similarly, since a file consisting of a large number of blocks may not fit into physical memory, a segment may not have enough blocks to reduce the decoding overhead. Finally, the decoding process is sensitive to the number of recovered degree-1 (unencoded) blocks that are generated with relatively low probability (e.g. 0.01). These blocks are necessary to start the decoding process and without a sufficient number of these blocks the decoding process cannot complete.

In the remainder of the paper, we optimistically assume that the entire file can fit into physical memory and quantify the potential benefits of using encoding at the source in Section 4.6.

¹We are assuming a memory-efficient implementation of the codes that releases the memory occupied by the encoded block when all the original blocks that were used in its construction are available. Performance can be even worse if the encoded blocks are kept until the file is reconstructed in full.

2.3 Peering Strategy

A node receives the file by peering with neighbors and receiving blocks from them. To enable this, the node requires techniques to learn about nodes in the system, selecting ones to peer with which have useful data, and determining the ideal set of peers which can optimize the incoming bandwidth of useful data. Ideally, a node would have perfect and timely information about distribution of blocks throughout the system and would be able to download any block from any other peer, but any such approach requiring global knowledge cannot scale. Instead, the system must approximate the peering decisions. It can do this by using a centralized coordination point, though constantly updating that point with updates of blocks received would also not scale, while also adding a single point of failure. Alternatively, a node could simply maintain a fixed set of peers, though this approach would suffer from changing network and node conditions or an initially poor selection of peers. One might also imagine using a DHT to coordinate location of nodes with given blocks. While we have not tested this approach, we reason that it would not perform well due to the extra overhead required for locating nodes with each block. Overall, a good approach to picking peers would be one which neither causes nodes to maintain global knowledge, nor communicate with a large number of nodes, but manages to locate peers which have a lot of data to give it. A good peering strategy will also allow the node to maintain a set of peers which is small enough to minimize control overhead, but large enough to keep the pipe full in the face of changing network conditions.

2.4 Request Strategy

In either a push or pull based system, there has to be a decision made about which blocks should be queued to send to which peers. In a pull based system, this is a request for block. Therefore we call this the request strategy. For the request strategy, we need to answer several important questions.

First, what is the best order for requesting blocks? For example, if all nodes make requests for the blocks in the same order, the senders in peering relationships will be sending the same data to all the peers, and there would be very little opportunity for nodes to help each other to speed up the overall progress.

Second, for any given block, more than one of the senders might have it. How does the node choose the sender that is going to provide this particular block, or in a pull based system, how can senders prevent queuing the same block for the same node? Further, should a node request the block immediately after it learns about its existence at a sender, or should it wait until some of its

other peers acquire the same block? There is a tradeoff, because reacting quickly might bring the block to this node sooner, and make it available for its own receivers to download sooner. However, the first sender over time that has this block might not be the best one to serve it; in this case it might be prudent to wait a bit longer, because the block download time from this sender might be high due to low available bandwidth.

Third, how much data should be requested from any given sender? Requesting too few blocks might not fill the node's incoming pipe, whereas requesting too much data might force the receiver to wait too long for a block that it could have requested and received from some other node.

Finally, where should the request logic be placed? One option is to have the receiver make explicit requests for blocks, which comes at the expense of maintaining data structures that describe the availability of each block, the time of requests, etc. In addition, this approach might incur considerable CPU overhead for choosing the next block to retrieve. If this logic is in the critical path, the throughput in high-bandwidth settings may suffer. Another option is to place the decision-making at the sender. This approach makes the receiver simpler, because it might just need to keep the sender up-to-date with a *digest* of blocks it currently has. Since a node might implicitly request the same block from multiple senders, this approach is highly resilient. On the other hand, duplicate blocks could be sent from multiple senders if senders do not synchronize their behavior. Further, message overhead will be higher than in the receiver-driven approach due to digests.

2.5 Sending Strategies

All techniques for distributing a file will need a strategy for sending data to peers to optimize the performance of the distribution. We define the sending strategy as "given a set of data items destined for a particular receiver, in what order should they be sent?" This is differentiated from the request strategy in that it is concerned with the *order* of queued blocks rather than *which* blocks to queue. Here, we separate the strategy of the single source from that of the many peers, since for any file distribution to be successful, the source must share all parts of the file with others.

Source

In this paper, we consider the case where the source is available to send file blocks for the entire duration of a file distribution session. This puts it in a unique position to be able to both help all nodes, and to affect the performance of the entire download. As a result, it is

amount of outstanding data, adjusting the overlay mesh over time to conform to the characteristics of the underlying network. Meanwhile, senders keep their receivers updated with the description of their newly received file blocks (step 6). The specifics of our implementation are described below.

3.2 Implementation

We have implemented Bullet' using MACEDON. MACEDON [22] is a tool which makes the development of large scale distributed systems simpler by allowing us to specify the overlay network algorithm without simultaneously implementing code to handle data transmission, timers, etc. The implementation of Bullet' consists of a generic file distribution application, the Bullet' overlay network algorithm, the existing basic random tree and RanSub algorithms, and the library code in MACEDON.

3.2.1 Download Application

The generic download application implemented for Bullet' can operate in either encoded or unencoded mode. In the encoded mode, it operates by generating continually increasing block numbers and transmitting them using `macedon_multicast` on the overlay algorithm. In unencoded mode, it operates by transmitting the blocks of the file once using `macedon_multicast`. This makes it possible for us to compare download performance over the set of overlays specified in MACEDON. In both encoded and unencoded cases, the download application also supports a request function from the overlay network to provide the block of a given sequence number, if available, for transmission. The download application also supports the reconstruction of the file from the data blocks delivered to it by the overlay. The download application is parameterized and can take as parameters block size and file name.

3.2.2 RanSub

RanSub is the protocol we use for distributing uniformly random subsets of peers to all nodes in the overlay. Unlike a centralized solution or one which requires state on the order of the size of the overlay, RanSub is decentralized and can scale better with both the number of nodes and the state maintained. RanSub requires an overlay tree to propagate random subsets, and in Bullet' we use the control tree for this purpose. RanSub works by periodically distributing a message to all members of the overlay, and then collecting data back up the tree. At each layer of the tree, data is randomized and compacted,

assuring that all peers see a changing, uniformly random subset of data. The period of this distribute and collect is configurable, but for Bullet', it is set for 5 seconds. Also, by decentralizing the random subset distribution, we can include more application-state, an abstract which nodes can use to make more intelligent decisions about which nodes would be good to peer with.

3.3 Algorithms

3.3.1 Peering Strategy

In order for nodes to fill their pipes with useful data, it is imperative that they are able to locate and maintain a set of peers that can provide them with good service. In the face of bandwidth changes and unstable network conditions, keeping a fixed number of peers is suboptimal (see Section 4.4). Not only must Bullet' discard peers whose service degrades, it must also adaptively decide how many peers it should be downloading from/sending to. Note that peering relationships are not inherently bidirectional; two nodes wishing to receive data from each other must establish peering links separately. Here we use the term "sender" to refer to a peer a node is receiving data from and "receiver" to refer to a peer a node is sending data to.

Each node maintains two variables, namely `MAX_SENDERS` and `MAX_RECEIVERS`, which specify how many senders and receivers the node wishes to have at maximum. Initially, these values are set to 10, the value we have experimentally chosen for the released version of Bullet. Bullet' also imposes hard limits of 6 and 25 for the number of minimum/maximum senders and receivers. Each time a node receives a RanSub distribute message containing a random subset of peers and the summaries of their file content, it makes an evaluation about its current peer sets and decides whether it should add or remove both senders and receivers. If the node has its current maximum number of senders, it makes a decision as to whether it should either "try out" a new connection or close a current connection based on the number of senders and bandwidth received when the last distribute message arrived. A similar mechanism is used to handle adding/removing receivers, except in this case Bullet' uses outgoing bandwidth instead of incoming bandwidth. Figure 2 shows the pseudocode for managing senders.

Once `MAX_SENDERS` and `MAX_RECEIVERS` have been modified, Bullet' calculates the average and standard deviation of bandwidth received from all of its senders. It then sorts the senders in order of least bandwidth received to most, and disconnects itself from any sender who is more than 1.5 standard deviations away from the mean, so long as it does not drop below the

```

void ManageSenders() {
    if (size(senders) != MAX_SENDERS)
        return;
    if (num_prev_senders == 0) {
        // try to add a new peer by default
        MAX_SENDERS++;
    }
    else if (size(senders) > num_prev_senders) {
        if (incoming_bw > prev_incoming_bw)
            // bandwidth went up, try adding
            // a sender
            MAX_SENDERS++;
        else
            // adding a new sender was bad
            MAX_SENDERS--;
    }
    else if (size(senders) < num_prev_senders) {
        if (incoming_bw > prev_incoming_bw)
            // losing a sender made us faster,
            // try losing another
            MAX_SENDERS--;
        else
            // losing a sender was bad
            MAX_SENDERS++;
    }
}

```

Figure 2: ManageSenders() Pseudocode

minimum number of connections (6). This way, Bullet' is able to keep only the peers who are the most useful to it. A fixed minimum bandwidth was not used so as to not hamper nodes who are legitimately slow. In addition, the slowest sender is not always closed since if all of a peer's senders are approximately equal in terms of bandwidth provided, then none of them should be closed.

A nearly identical procedure is executed to remove receivers who are potentially limiting the outgoing bandwidth of a node. However, Bullet' takes care to sort receivers based on the ratio of their bandwidth they are receiving from a particular sender to their total incoming bandwidth. This is important because we do not want to close peers who are getting a large fraction of their bandwidth from a given sender. We chose the value of 1.5 standard deviations because 1 would lead to too many nodes being closed whereas 2 would only permit a very few peers to ever be closed.

3.3.2 Request Strategy

We considered using four different request strategies when designing Bullet'. All of the strategies are for making local decisions on either the unencoded or source-encoded file. Given a per-peer list representing blocks that are available from that peer, the following are possible ways to order requests:

- **First encountered** This strategy will simply arrange the lists based on block availability. That is, blocks that are just discovered by a node will be requested after blocks the node has known about for

a while. As an example, this might correspond to all nodes proceeding in lockstep in terms of download progress. The resulting low block diversity this causes in the system could lead to lower performance.

- **Random** This method will randomly order each list with the intent of improving the block diversity in the system. However, there is a possibility of requesting a block that many other nodes already have which does *not* help block diversity. As a result, this strategy might not significantly improve the overall system performance.
- **Rarest** The *rarest* technique is the first that looks at block distributions among a node's peers when ordering the lists. Each list will be ordered with the least represented blocks appearing first. This strategy has no method for breaking ties in terms of rarity, so it is possible that blocks quickly go from being under represented to well represented when a set of nodes makes the same deterministic choice.
- **Rarest random** The final strategy we describe is an improvement over the *rarest* approach in that it will choose uniformly at random from the blocks that have the highest rarity. This strategy eliminates the problem of deterministic choices leading to suboptimal conditions.

In order to decide which strategy worked the best, we implemented all four in Bullet'. We present our findings in Section 4.3.

3.3.3 Flow Control

Although the *rarest random* request strategy enables Bullet' to request blocks from peers in a way that encourages block diversity, it does not specify how many blocks a node should request from its peers at once. This choice presents a tradeoff between control overhead (making requests) and adaptivity. On one hand, a node could request one block at a time, not requesting another one until the first arrived. Although stopping and waiting would provide maximum insulation to changing network conditions, it would also leave pipes underutilized due to the round trip time involved in making the next request. At the other end of the spectrum is a strategy where a node would request everything that it knew about from its peers as soon as it learned about it. In this case, the control traffic is reduced and the node's pipe from each of its peers has a better chance of being full but this technique has major flaws when network conditions change. If a peer suddenly slows down, the node will find itself stuck waiting for a large number of blocks to

```

void ManageOutstanding (sender, block) {
// start with current value
sender->desired = sender->requested + 1;

if (block->wasted <= 0 || block->in_front <= 1)
    sender->desired -=
        0.4*block->wasted*sender->bandwidth/block_size);

if (block->wasted <= 0 && block->in_front > 1)
    sender->desired -= 0.226*(block->in_front - 1);
}

```

Figure 3: Pseudocode for setting the maximum number of per-peer outstanding blocks.

come in at a slow rate. We have experimented with canceling of blocks that arrive “slowly”, and found that in many cases these blocks are “in-flight” or in the sender’s socket buffer, making it difficult to effectively stop their retrieval without closing the TCP connection.

As seen in Section 4.5, using a fixed number of outstanding blocks will not perform well under a wide variety of conditions. To remedy this situation, Bullet’ employs a novel flow control algorithm that attempts to dynamically change the maximum number of blocks a node is willing to have outstanding from each of its peers. Our control algorithm is similar to XCP’s [11] efficiency controller, the feedback control loop for calculating the aggregate feedback for all the flows traversing a link. XCP measures the difference in the rates of incoming and outgoing traffic on a link, and computes the total number of bytes by which the flows’ congestion windows should increase or decrease. XCP’s goal is to maintain 0 packets queued on the bottleneck link. For the particular values of control parameters $\alpha = 0.4, \beta = 0.226$, the control loop is stable for any link bandwidth and delay.

We start with the original XCP formula and adapt it. Since we want to keep each pipe full while not risking waiting for too much data in case the TCP connection slows down, our goal is to maintain exactly 1 block in front of the TCP socket buffer, for each peer. With each block it sends, sender measures and reports two values to the receiver that runs the algorithm depicted in Figure 3 in the pseudocode. The first value is `in_front`, corresponding to the number of queued blocks in front of the socket buffer when the request for the particular block arrives. The second value is `wasted`, and it can be either positive or negative. If it is negative, it corresponds to the time that is “wasted” and could have been occupied by sending blocks. If it is positive, it represents the “service” time this block has spent waiting in the queue. Since this time includes the time to service each of the `in_front` blocks, we take care not to double count the service time in this case. To convert the wasted (service) time into units applicable to the formula, we multiply it by the bandwidth measured at the receiver, and divide by

block size to derive the additional (reduced) number of blocks receiver could have requested. Once we decide to change the number of outstanding blocks, we mark a block request and do not make any adjustments until that block arrives. This technique allows us to observe any changes caused by our control algorithm before taking any additional action. Further, just matching the rate at which the blocks are requested with the sending bandwidth in an XCP manner would not saturate the TCP connection. Therefore, we take the `ceiling` of the non-integer value for the desired number of outstanding blocks whenever we increase this value.

Although Bullet’ knows how much data it should request and from whom, a mechanism is still needed that specifies when the requests should be made. Initially, the number of blocks outstanding for all peers starts at 3, so when a node gains a sender it will request up to 3 blocks from the new peer. Conceptually, this corresponds to the pipeline of one block arriving at the receiver, with one more in-flight, and the request for the third block reaching the sender. Whenever a block is received, the node re-evaluates the potential from this peer and requests up to the new maximum outstanding.

3.3.4 Staying Up-To-Date

One small detail we have deferred until now is how nodes become aware of what their peers have. Bullet’ nodes use a simple bitmap structure to transmit *diffs* to their peers. These diffs are incremental, such that a node will only hear about a particular block from a peer once. This approach helps to minimize wasted bandwidth and decouples the size of the diff from the size of the file being distributed. Currently, a diff may be transmitted from node A to B in one of two circumstances - either because B has nothing requested of A, or because B specifically asked for a diff to be sent. The latter would occur when B is about to finish requesting all of the blocks A currently has. An interesting effect of this mechanism is that diff sending is automatically self clocking; there are no fixed timers or intervals where diffs are sent at a specific rate. Bullet’ automatically adjusts to the data consumption rates of each individual peer.

3.3.5 Sending Strategy

As mentioned previously, Bullet’ uses a hybrid push/pull approach for data distribution where the source behaves differently from everyone else. The source takes a rather simple approach: it sends a block to each of its RanSub children iteratively until the entire file has been sent. If a block cannot be sent to one child (because the pipe to it is already full), the source will try the next child in a round robin fashion until a suitable recipient is found. In

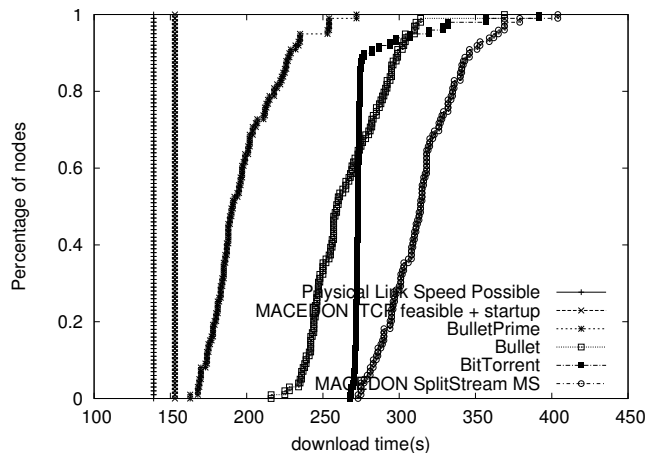


Figure 4: Comparison of Bullet' to BitTorrent, Bullet, SplitStream and optimal case for 100MB file size under random network packet losses. The legend is ordered top to bottom, while the lines are ordered from left to right.

this manner, the source never wastes bandwidth forcing a block on a node that is not ready to accept it. Once the source makes each of the file blocks available to the system, it will advertise itself in RanSub so that arbitrary nodes can benefit from having a peer with all of the file blocks.

From the perspective of non-source nodes, determining the order in which to send requested blocks is equally as simple. Since Bullet' dynamically determines the number of outstanding requests, nodes should always have approximately one outstanding request at the application level on any peer at any one time. As a result, the sender can simply serve requests in FIFO order since there is not much of a choice to make among such few blocks. Note that this approach would not be optimal for all systems, but since Bullet' dynamically adjusts the number of requests to have outstanding for each peer, it works well.

4 Evaluation

To conduct a rigorous analysis of our various design tradeoffs, we needed to construct a controlled experimental environment where we could conduct multiple tests under identical conditions. Rather than attempt to construct a necessarily small isolated network test-bed, we present results from experiments using the ModelNet [28] network emulator, which allowed us to evaluate Bullet' on topologies consisting of 100 nodes or more. In addition, we present experimental results over the wide-area network using the PlanetLab [20] testbed.

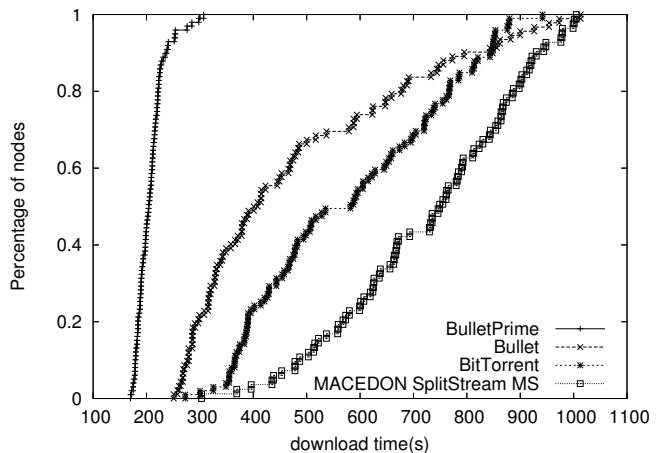


Figure 5: Comparison of Bullet' to BitTorrent, Bullet, and SplitStream for 100MB file size under synthetic bandwidth changes and random network packet losses.

4.1 Experimental Setup

Our ModelNet experiments make use of 25 2.0 and 2.8-GHz Pentium-4s running Xeno-Linux 2.4.27 and interconnected by 100-Mbps and 1-Gbps Ethernet switches. In the experiments presented here, we multiplex one hundred logical end nodes running our download applications across the 25 Linux nodes (4 per machine). ModelNet routes packets from the end nodes through an emulator responsible for accurately emulating the hop-by-hop delay, bandwidth, and congestion of a given network topology; a 1.4-GHz Pentium III running FreeBSD-4.7 served as the emulator for these experiments.

All of our experiments are run on a fully interconnected mesh topology, where each pair of overlay participants are directly connected. While admittedly not representative of actual Internet topologies, it allows us maximum flexibility to affect the bandwidth and loss rate between any two peers. The inbound and outbound access links of each node are set to 6 Mbps, while the nominal bandwidth on the core links is 2 Mbps. In an attempt to model the wide-area environment [21], we configure ModelNet to randomly drop packets on the core links with probability ranging from 0 to 3 percent. The loss rate on each link is chosen uniformly at random and fixed for the duration of an experiment. To approximate the latencies in the Internet [7, 21], we set the propagation delay on the core links uniformly at random between 5 and 200 milliseconds, while the access links have one millisecond delay.

For most of the following sections, we conduct identical experiments in two scenarios: a static bandwidth case and a variable bandwidth case. Our bandwidth-change scenario models changes in the network band-

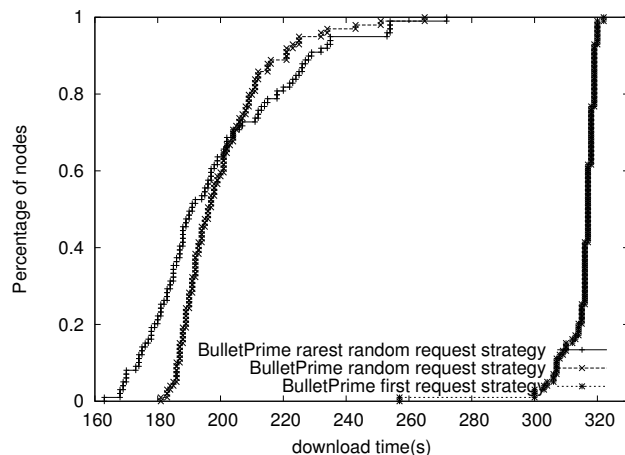


Figure 6: Impact of request strategy on Bullet' performance while downloading a 100 MB file under random network packet losses.

width that correspond to correlated and cumulative decreases in bandwidth from a large set of sources from any vantage point. To effect these changes, we decrease the bandwidth in the core links with a period of 20 seconds. At the beginning of each period, we choose 50 percent of the overlay participants uniformly at random. For each participant selected, we then randomly choose 50 percent of the other overlay participants and decrease the bandwidth on the core links from those nodes to 50 percent of the current value, without affecting the links in the reverse direction. The changes we make are cumulative; i.e., it is possible for an unlucky node pair to have 25% of the original bandwidth after two iterations. We do not alter the physical link loss rates that were chosen during topology generation.

4.2 Overall Performance

We begin by studying how Bullet' performs overall, using the existing best-of-breed systems as comparison points. For reference, we also calculate the best achievable performance given the overhead of our underlying transport protocols. Figure 4 plots the results of downloading a 100-MB file on our ModelNet topology using a number of different systems. The graph plots the cumulative distribution function of node completion times for four experimental runs and two calculations. Starting at the left, we plot download times that are optimal with respect to access link bandwidth in the absence of any protocol overhead. We then estimate the best possible performance of a system built using MACEDON on top of TCP, accounting for the inherent delay required for nodes to achieve maximum download rate. The remaining four lines show the performance of Bullet' running in

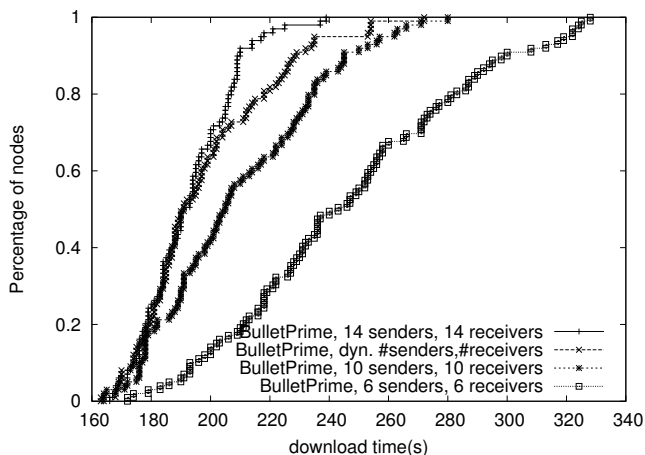


Figure 7: Bullet' performance under random network packet losses for static peer set sizes of 6, 10, 14, and the dynamic peer set size sizing case while downloading a 100 MB file.

the unencoded mode, Bullet, and BitTorrent, our MACEDON SplitStream implementation, in roughly that order. Bullet' clearly outperforms all other schemes by approximately 25%. The slowest Bullet' receiver finishes downloading 37% faster than for other systems. Bullet's performance is even better in the dynamic scenario (faster by 32%-70%), shown in Figure 5.

We set the transfer block size to 16 KB in all of our experiments. This value corresponds to BitTorrent's sub-piece size of 16KB, and is also shared by the Bullet and SplitStream. For all of our experiments, we make sure that there is enough physical memory on the machines hosting the overlay participants to cache the entire file content in memory. Our goal is to concentrate on distributed algorithm performance and not worry about swapping file blocks to and from the disk. Bullet and SplitStream results are optimistic since we do not perform encoding and decoding of the file. Instead, we set the encoding overhead to 4% and declare the file complete when a node has received enough file blocks.

4.3 Request Strategy

Heartened by the performance of Bullet' with respect to other systems, we now focus our attention on the various critical aspects of our design that we believe contribute to Bullet's superior performance. Figure 6 shows the performance of Bullet' using three different peer request strategies, again using the CDF of node completion times. In this case each node is downloading a 100 MB file. We argue the goal of a request strategy is to promote block diversity in the system, allowing nodes to help each other. Not surprisingly, we see that the *first-encountered*

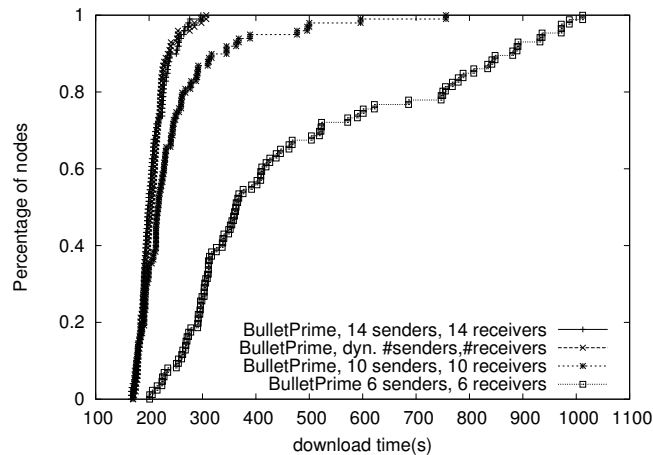


Figure 8: Bullet' performance under synthetic bandwidth changes and random network packet losses for static peer set sizes of 6, 10, 14, and the dynamic peer set size sizing case while downloading a 100 MB file.

request strategy performs the worst. While the rarest-random performs best among the strategies considered for 70% of the receivers. For the slowest nodes, the random strategy performs better. When a receiver is downloading from senders over lossy links, higher loss rates increase the latency of block availability messages due to TCP retransmissions and use of the congestion avoidance mechanism. Subsequently, choosing the next block to download uniformly at random does a better job of improving diversity than the rarest-random strategy that operates on potentially stale information.

4.4 Peer Selection

In this section we demonstrate the impossibility of choosing a single optimal number of senders and receivers for each peer in the system, arguing for a dynamic approach. In Figure 7 we contrast Bullet's performance with 10 and 14 peers (for both senders and receivers) while downloading a 100 MB file. The system configured with 14 peers outperforms the one with 10 because in a lossy topology like the one we are using, having more TCP flows makes the node's incoming bandwidth more resilient to packet losses. Our dynamic approach is configured to start with 10 senders and receivers, but it closely tracks the performance of the system with the number of peers fixed to 14 for 50% of receivers. Under synthetic bandwidth changes (Figure 8), our dynamic approach matches, and sometimes exceeds the performance of static setups.

For our final peering example, we construct a 100 node topology with ample bandwidth in the core (10Mbps, 1ms latency links) with 800 Kbps access links and with-

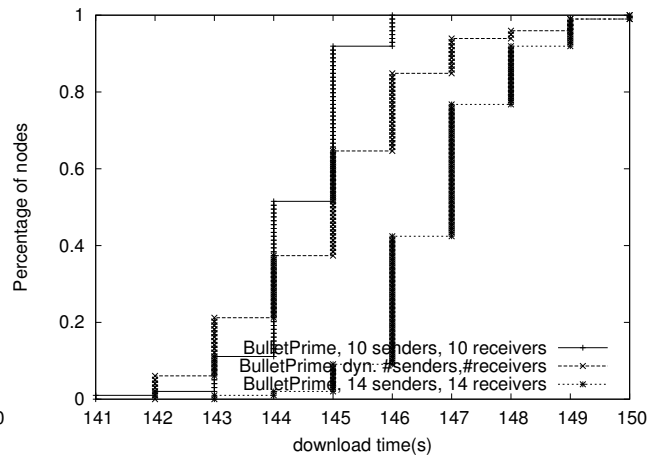


Figure 9: Bullet' performance in the absence of bandwidth changes and random packet losses for static peer set sizes of 10, 14, and the dynamic peer set size sizing case while downloading a 10 MB file in a topology with constrained access links.

out random network packet losses. Figure 9 shows that, unlike in the previous experiments, Bullet' configured for 14 peers performs *worse* than in a setup with 10 peers. Having more peers in this constrained environment forces more maximizing TCP connections to compete for bandwidth. In addition, maintaining more peers requires sending more control messages, further decreasing the system performance. Our dynamic approach tracks, and sometimes exceeds, the performance of the better static setup.

These cases clearly demonstrate that no statically configured peer set size is appropriate for a wide range of network environments, and a well-tuned system must dynamically determine the appropriate peer set size.

4.5 Outstanding Requests

We now explore determining the optimal number of peer outstanding requests. Other systems use a fixed number of outstanding blocks. For example, BitTorrent tries to maintain five outstanding blocks from each peer by default. For the experiments in this section, we use an 8KB block, and configure the Linux kernel to allow large receiver window sizes. In our first topology, there are 25 participants, interconnected with 10Mbps links with 100ms latency. In Figure 10 we show Bullet's performance when configured with 3, 6, 9, 15, and 50 peer outstanding blocks for up to 5 senders. The number of outstanding requests refers to the *total* number of block requests to any given peer, including blocks that are queued for sending, and blocks and requests that are in-flight. As we can see, the dynamic technique closely

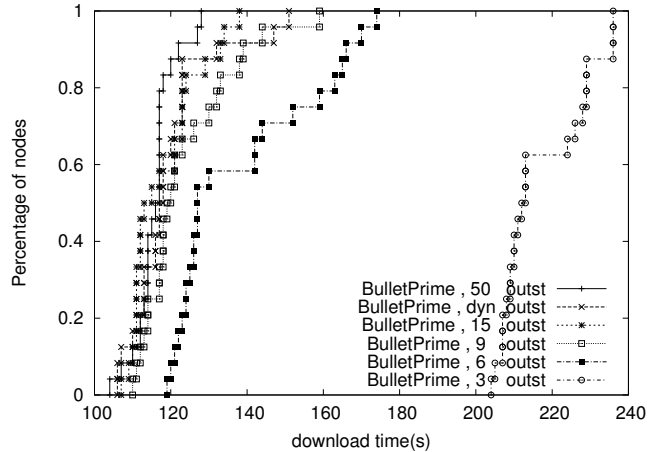


Figure 10: Bullet' performance with neither bandwidth changes nor random network packet losses for 3, 6, 9, 15, 50 outstanding blocks, and for the dynamic queue sizing case while downloading a 100 MB file

tracks the performance of cases with a large number of outstanding blocks. Having too few outstanding requests is not enough to fill the bandwidth-delay product of high-bandwidth, high-latency links.

Although it is tempting to simplify the system by requesting the maximum number of blocks from each peer, Figure 11 illustrates the penalty of requesting more blocks than it is required to saturate the TCP connection. In this experiment, we instruct ModelNet to drop packets uniformly at random with probability ranging between 0 and 1.5 percent on the core links. Due to losses, TCP achieves lower bandwidths, requiring less data in-flight for maximum performance. Under these loss-induced TCP throughput fluctuations, our dynamic approach outperforms all static cases. Figure 12 provides more insight into this case. For this experiment, we have 8 participants including the source, with 6 nodes receiving data from the source and reconciling among each other over 10Mbps, 1ms latency links. We use 8KB blocks and disable peer management. The 8th node is only downloading from the 6 peers over dedicated 5Mbps, 100ms latency links. Every 25 seconds, we choose another peer from these 6, and reduce the bandwidth on its link toward the 8th node to 100Kbps. These cascading bandwidth changes are cumulative, i.e. in the end, the 8th node will have only 100Kbps links from its peers. Our dynamic scheme outperforms all fixed sizing choices for the slowest, 8th node, by 7 to 22 percent. Placing too many requests on a connection to a node that suddenly slows down forces the receiver to wait too long for these blocks to arrive, instead of retrieving them from some other peer.

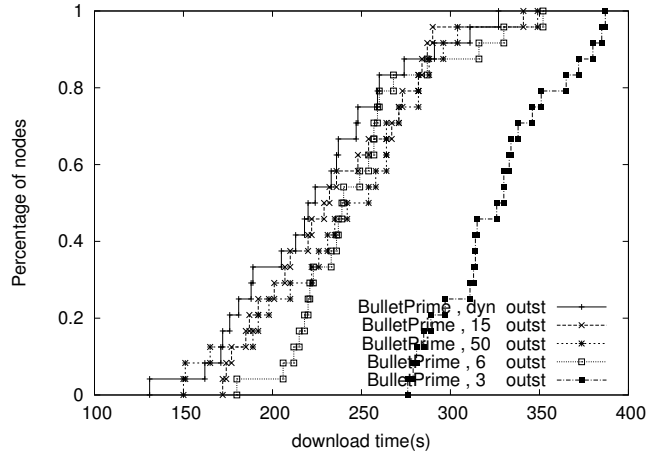


Figure 11: Bullet' performance under random network packet losses while downloading a 100 MB file. CDF line for 9 blocks omitted for readability.

4.6 Potential Benefits of Source Encoding

The purpose of this section is to quantify the potential benefits of encoding the file at the source. Towards this end, Figure 13 shows the average block inter-arrival times among the 99 receivers, while downloading a 100MB file. To improve the readability of the graph, we do not show the maximum block inter-arrival times, which observe a similar trend. A system that has a pronounced “last-block” problem would exhibit a sharp increase in the block inter-arrival time for the last several blocks. To quantify the potential benefits of encoding, we first compute the overall average block inter-arrival time t_b . We then consider the last twenty blocks and calculate the cumulative overage of the average block inter-arrival time over t_b . In this case overage amounts to 8.38 seconds. We contrast this value to the potential increase in the download time due to a fixed 4 percent encoding overhead of 7.60 seconds, while optimistically assuming that downloads using source encoding would not exhibit any deviation in the download times of the last few blocks. We conclude that encoding at the source in this scenario would not be of clear benefit in improving the average download time. This finding can be explained by the presence of a large number of nodes that will have a particular block and will be available to send it to other participants. Encoding at the source or within the network can be useful when the source becomes unavailable soon after sending the file once and with node churn [8].

4.7 PlanetLab

This section contains results from the deployment of Bullet' over the PlanetLab [20] wide-area network

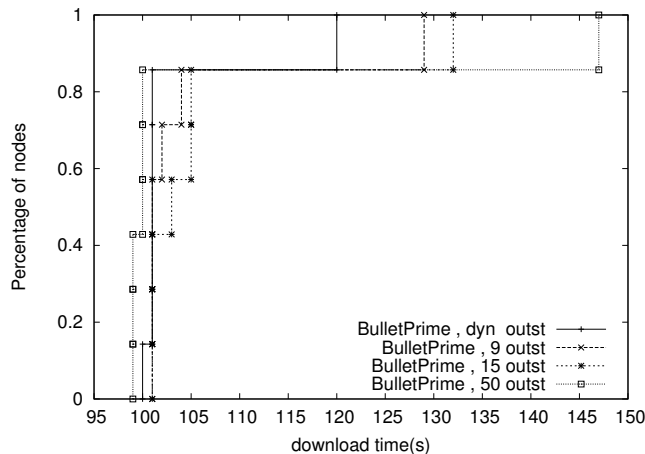


Figure 12: Bullet' performance under synthetic bandwidth changes while downloading a 100 MB file. CDF lines for 3 and 6 omitted because the 8th node takes considerably more time to finish in these cases.

testbed. For our first experiment, we chose 41 nodes for our deployment, with no two machines being deployed at the same site. We configured Bullet', Bullet, and SplitStream (MACEDON MS implementation) to use a 100KB block size. Bullet and SplitStream were not performing the file encoding/decoding; instead we mark the downloads as successful when a required number of distinct file blocks is successfully received, including fixed 4 percent overhead that an actual encoding scheme would incur. We see in Figure 14 that Bullet' consistently outperforms other systems in the wide-area. For example, the slowest Bullet' node completes the 50MB download approximately 400 seconds sooner than BitTorrent's slowest downloader.

4.8 Shotgun: a Rapid Synchronization Tool

This section presents Shotgun, a set of extensions to the popular `rsync` [27] tool to enable clients to synchronize their state with a centralized server orders of magnitude more quickly than previously possible. At a high-level, Shotgun works by wrapping appropriate interfaces to Bullet' around `rsync`. Shotgun is useful, for example, for a user who wants to run an experiment on a set of PlanetLab[20] nodes. Because each node only has local storage, the user must somehow copy (using `scp` or `rsync`) her program files to each node individually. Each time she makes any change to her program, she must re-copy the updated files to all the nodes she is using. If the experiment spans hundreds of nodes, then copying the files requires opening hundreds of ssh connections, all of which compete for bandwidth.

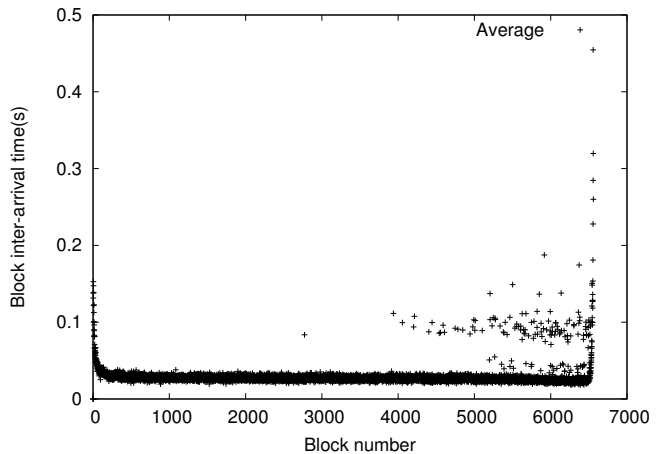


Figure 13: Block inter-arrival times for a 100 MB file under random network packet losses in the absence of bandwidth changes. The block numbers on the X axis correspond to the order in which nodes retrieve blocks, not the actual block numbers.

To use Shotgun, the user simply starts `shotgund`, the Shotgun multicast daemon, on each of his nodes. To distribute an update, the user runs `shotgun_sync`, providing as arguments a path to the new software image, a path to the current software image, and the host name of the source of the Shotgun multicast tree (this can be the local machine). Next, `shotgun_sync` runs `rsync` in batch mode between the two software image paths, generating a set of logs describing the differences and records the version numbers of the old and new files. `shotgun_sync` then archives the logs into a single `tar` file and sends it to the source, which then rapidly disseminates it to all the clients using the multicast overlay. Each client's `shotgund` will download the update, and then invoke `shotgun_sync` locally to apply the update if the update's version is greater than the node's current version.

Running an `rsync` instance for each target node overloads the source node's CPU with a large number of `rsync` processes all competing for the disk, CPU, and bandwidth. Therefore, we have attempted to experimentally determine the number of simultaneous `rsync` processes that give the optimal overall performance using the staggered approach. Figure 15 shows that Shotgun outperforms `rsync` (1, 2, 4, 8, and 16 parallel instances) by two orders of magnitude. Another interesting result from this graph is that the constraining factor for PlanetLab nodes is the disk, not the network. On average, most nodes spent twice as much time replaying the `rsync` logs locally then they spent downloading the data.

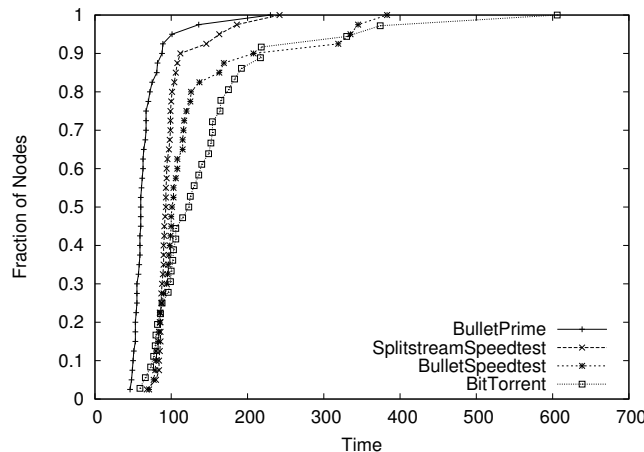


Figure 14: Comparison of Bullet' to Bullet, BitTorrent, and SplitStream for 50MB file size on PlanetLab.

5 Related Work

Overcast [10] constructs a bandwidth-optimized overlay tree among dedicated infrastructure nodes. An incoming node joins at the source and probes for acceptable bandwidth under one if its siblings to descend down the tree. A node's bandwidth and reliability is determined by characteristics of the network between itself and its parent and is lower than the performance and reliability provided by an overlay mesh.

Narada [9] constructs a mesh based on a k -spanner graph, and uses bandwidth and latency probing to improve the quality of the mesh. It then employs standard routing algorithms to compute per-source forwarding trees, in which a node's performance is still defined by connectivity to its parent. In addition, the group membership protocol limits the scale of the system.

Snoeren et al. [26] use an overlay mesh to send XML-encoded data. The mesh is structured by enforcing k parents for each participant. The emphasis of this primarily push-based system is on reliability and timely delivery, so nodes flood the data over the mesh.

In FastReplica [6] file distribution system, the source of a file divides the file into n blocks, sends a different block to each of the receivers, and then instructs the receivers to retrieve the blocks from each other. Since the system treats every node pair equally, overall performance is determined by the transfer rate of the slowest end-to-end connection.

BitTorrent [3] is a system in wide use in the Internet for distribution of large files. Incoming nodes rely on the centralized *tracker* to provide a list of existing system participants and system-wide block distribution for random peering. BitTorrent enforces fairness via a tit-for-tat mechanism based on bandwidth. Our inspection

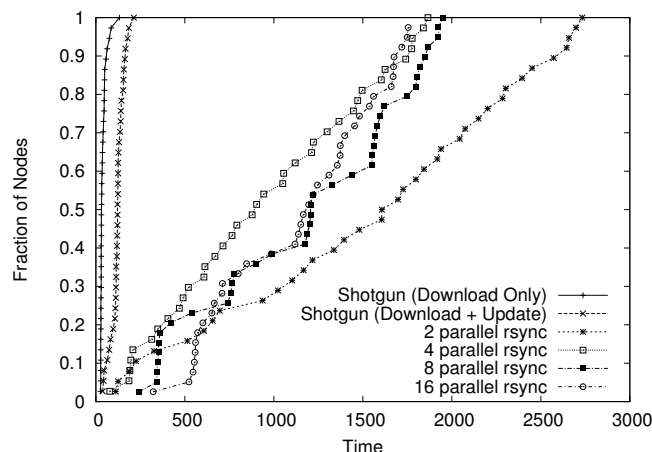


Figure 15: The aggregate completion time for varying number of parallel rsync processes for an update with 24MB of deltas on 40 PlanetLab nodes.

of the BitTorrent code reveals hard coded constants for request strategies and peering strategies, potentially limiting the adaptability of the system to a variety of network conditions relative to our approach. In addition, tracker presents a single point of failure and limits the system scalability.

SplitStream [5] aims to construct an interior-node disjoint forest of k Scribe [24] trees on top of a scalable peer-to-peer substrate [23]. The content is split into k stripes, each of which is pushed along one of the trees. This system takes into account physical inbound and outbound link bandwidth of a node when determining the number of stripes a node can forward. It does not, however, consider the overall end-to-end performance of an overlay path. Therefore, system throughput might be decreased by congestion that does not occur on the access links.

In CoopNet [18], the source of the multimedia content computes locally random or node-disjoint forests of trees in a manner similar to SplitStream. The trees are primarily designed for resilience to node departures, with network efficiency as the second goal.

Slurpie [25] improves upon the performance of BitTorrent by using an adaptive downloading mechanism to scale the number of peers a node should have. However, it does not have a way of dynamically changing the number of outstanding blocks on a per-peer basis. In addition, although Slurpie has a random backoff algorithm that prevents too many nodes from going to the source simultaneously, nodes can connect to the webserver and request arbitrary blocks. This would increase the minimum amount of time it takes all blocks to be made available to the Slurpie network, hence leading to increased

minimum download time.

Avalanche [8] is a file distribution system that uses network coding [1]. The authors demonstrate the usefulness of producing encoded blocks by all system participants under scenarios when the source departs soon after sending the file once, and on specific network topologies. There are no live evaluation results of the system, but it is likely Avalanche will benefit from the techniques outlined in this paper. For example, Avalanche participants will have to choose a number of sending peers that will fill their incoming pipes. In addition, receivers will have to negotiate carefully the transfers of encoded blocks produced at random to avoid bandwidth waste due to blocks that do not aid in file reconstruction, while keeping the incoming bandwidth high from each peer.

CoDeploy [19] builds upon CoDeeN, an existing HTTP Content Distribution Network (CDN), to support dissemination of large files. In contrast, Bullet' operates without infrastructure support and achieves bandwidth rates (7Mbps on average with a source limited to 10Mbps) that exceed CoDeploy's published results.

Young et al. [29] construct an overlay mesh of k edge-disjoint minimum cost spanning trees (MSTs). The algorithm for distributed construction of trees uses overlay link metric information such as latency, loss rate, or bandwidth that is determined by potentially long and bandwidth consuming probing stage. The resulting trees might start resembling a random mesh if the links have to be excluded in an effort to reduce the probing overhead. In contrast, Bullet' builds a content-informed mesh and completely eliminates the need for probing because it uses transfers of useful information to adapt to the characteristics of the underlying network.

6 Conclusions

We have presented Bullet', a system for distributing large files across multiple wide-area sites in a wide range dynamic of network conditions. Through a careful evaluation of design space parameters, we have designed Bullet' to keep its incoming pipe full of useful data with a minimal amount of control overhead from a dynamic set of peers. In the process, we have defined the important aspects of the general data dissemination problem and explored several possibilities within each aspect. Our results validate that each of these tenets of file distribution is an important consideration in building file distribution systems. Our experience also shows that strategies which have tunable parameters might perform well in a certain range of conditions, but that once outside that range they will break down and perform worse than if they had been tuned differently. To combat this problem, Bullet' employs adaptive strategies which can adjust over time to self-tune to conditions which will perform well in a much

wider range of conditions, and indeed in many scenarios of dynamically changing conditions. Additionally, we have compared Bullet' with BitTorrent, Bullet and SplitStream. In all cases, Bullet' outperforms other systems.

Acknowledgments

We would like to thank David Becker for his invaluable help with our ModelNet experiments and Ken Yocum for his comments on our flow control algorithm. In addition, we thank our shepherd Atul Adya and our anonymous reviewers who provided excellent feedback.

References

- [1] Rudolf Ahlswede, Ning Cai, Shuo-Yen Robert Li, and Raymond W. Yeung. Network Information Flow. In *IEEE Transactions on Information Theory*, vol. 46, no. 4, 2000.
- [2] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable Application Layer Multicast. In *Proceedings of ACM SIGCOMM*, August 2002.
- [3] Bittorrent. <http://bitconjurer.org/BitTorrent>.
- [4] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *Proceedings of ACM SIGCOMM*, 1998.
- [5] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth Content Distribution in Cooperative Environments. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.
- [6] Ludmila Cherkasova and Jangwon Lee. FastReplica: Efficient Large File Distribution within Content Delivery Networks. In *4th USENIX Symposium on Internet Technologies and Systems*, March 2003.
- [7] Frank Dabek, Jinyang Li, Emil Sit, Frans Kaashoek, Robert Morris, and Chuck Blake. Designing a dht for low latency and high throughput. In *Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, March 2004.
- [8] Christos Gkantsidis and Pablo Rodriguez Rodriguez. Network Coding for Large Scale Content Distribution. In *Proceedings of IEEE Infocom*, 2005.
- [9] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of ACM SIGCOMM*, August 2001.
- [10] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and Jr. James W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, October 2000.

- [11] Dina Katabi, Mark Handley, and Charles Rohrs. Internet congestion control for high bandwidth-delay product networks. In *Proceedings of ACM SIGCOMM*, August 2002.
- [12] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using Random Subsets to Build Scalable Network Services. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, March 2003.
- [13] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High Bandwidth Data Dissemination Using and Overlay Mesh. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.
- [14] Maxwell N. Krohn, Michael J. Freedman, and David Mazieres. On-the-Fly Verification of Rateless Erasure Codes for Efficient Content Distribution. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 2004.
- [15] Michael Luby. LT Codes. In *In The 43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002.
- [16] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical Loss-Resilient Codes. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC '97)*, pages 150–159, New York, May 1997. Association for Computing Machinery.
- [17] Petar Maymounkov and David Mazieres. Rateless codes and big downloads. In *Proceedings of the Second International Peer to Peer Symposium (IPTPS)*, March 2003.
- [18] Venkata N. Padmanabhan, Helen J. Wang, and Philip A. Chou. Resilient Peer-to-Peer Streaming. In *Proceedings of the 11th ICNP*, Atlanta, Georgia, USA, 2003.
- [19] KyoungSoo Park and Vivek S. Pai. Deploying large file transfer on an http content distribution network. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS '04)*, 2004.
- [20] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of ACM HotNets-I*, October 2002.
- [21] Pinger site-by-month history table. <http://www.iepm.slac.stanford.edu/cgi-wrap/pingtable.pl>.
- [22] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, and Amin Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, March 2004.
- [23] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Middleware'2001*, November 2001.
- [24] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The Design of a Large-scale Event Notification Infrastructure. In *Third International Workshop on Networked Group Communication*, November 2001.
- [25] Rob Sherwood, Ryan Braud, and Bobby Bhattacharjee. Slurpie: A Cooperative Bulk Data Transfer Protocol. In *Proceedings of IEEE INFOCOM*, 2004.
- [26] Alex C. Snoeren, Kenneth Conley, and David K. Gifford. Mesh-Based Content Routing Using XML. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.
- [27] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, 1999.
- [28] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.
- [29] Anthony Young, Jiang Chen, Zheng Ma, Arvind Krishnamurthy, Larry Peterson, and Randolph Y. Wang. Overlay mesh construction using interleaved spanning trees. In *Proceedings of IEEE INFOCOM*, 2004.

Server Network Scalability and TCP Offload

Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz,
Erich Nahum, Prashant Pradhan, John Tracey
IBM T. J. Watson Research Center
Hawthorne, NY, 10532

{dmfreim, elbert, lavoie, mraz, nahum, ppradhan, traceyj}@us.ibm.com

Abstract

Server network performance is increasingly dominated by poorly scaling operations such as I/O bus crossings, cache misses and interrupts. Their overhead prevents performance from scaling even with increased CPU, link or I/O bus bandwidths. These operations can be reduced by redesigning the host/adaptor interface to exploit additional processing on the adaptor. Offloading processing to the adaptor is beneficial not only because it allows more cycles to be applied but also of the changes it enables in the host/adaptor interface. As opposed to other approaches such as RDMA, TCP offload provides benefits without requiring changes to either the transport protocol or API.

We have designed a new host/adaptor interface that exploits offloaded processing to reduce poorly scaling operations. We have implemented a prototype of the design including both host and adaptor software components. Experimental evaluation with simple network benchmarks indicates our design significantly reduces I/O bus crossings and holds promise to reduce other poorly scaling operations as well.

1 Introduction

Server network throughput is not scaling with CPU speeds. Various studies have reported CPU scaling factors of 43% [23], 60% [15], and 33% to 68% [22] which fall short of an ideal scaling of 100%. In this paper, we show that even increasing CPU speeds and link and bus bandwidths does not generate a commensurate increase in server network throughput. This lack of scalability points to an increasing tendency for server network throughput to become the key bottleneck limiting system performance. It motivates the need for an alternative design with better scalability.

Server network scalability is limited by operations heavily used in current designs that themselves do not scale well, most notably bus crossings, cache misses and interrupts. Any significant improvement in scalability must reduce these operations. Given that the problem is one of scalability and not simply performance, it will not be solved by faster processors. Faster processors merely

expend more cycles on poorly scaling operations.

Research in server network performance over the years has yielded significant improvements including: integrated checksum and copy, checksum offload, copy avoidance, interrupt coalescing, fast path protocol processing, efficient state lookup, efficient timer management and segmentation of offload, a.k.a. large send. Another technique, full TCP offload, has been pursued for many years. Work on offload has generated both promising and less than compelling results [1, 38, 40, 42]. Good performance data and analysis on offload is scarce.

Many improvements in server scalability were described more than fifteen years ago by Clark et al. [9]. The authors demonstrated that the overhead incurred by network protocol processing, per se, is small compared to both per-byte (memory access) costs and operating system overhead, such as buffer and timer management. This motivated work to reduce or eliminate data touching operations, such as copies, and to improve the efficiency of operating system services heavily used by the network stack. Later work [19] showed that overhead of non-data touching operations is, in fact, significant for real workloads, which tend to feature a preponderance of small messages. Today, per-byte overhead has been greatly reduced through checksum offload and zero-copy send. This leaves per-packet overhead, operating system services and zero-copy receive as the main remaining areas for further improvement.

Nearly all of the enhancements described by Clark et al. have seen widespread adoption. The one notable exception is an efficient network interface. This is a network adaptor with a fast general-purpose processor that provides a much more efficient interface to the network than the current frame-based interface devised decades ago. In this paper, we describe an effort to develop a much more efficient network interface and to make this enhancement a reality as well.

Our work is pursued in the context of TCP for three reasons: 1) TCP's enormous installed base, 2) the methodology employed with TCP will transfer to other protocols, and 3) the expectation that key new architectural features, such as zero copy receive, will ultimately demonstrate their viability with TCP.

The work described here is part of a larger effort to improve server network scalability. We began by analyzing server network performance and recognizing, as others have, a significant scalability problem. Next, we identified specific operations to be the cause, specifically: bus crossings, cache misses, and interrupts. We formulated a design that reduces the impact of these operations. This design exploits additional processing at the network adapter, i.e. offload, to improve the efficiency of the host/adapter interface which is our primary focus. We have implemented a prototype of the new design which consists of host and adapter software components and have analyzed the impact of the new design on bus crossings. Our findings indicate that offload can substantially decrease bus crossings and holds promise to reduce other scalability limiting operations such as cache misses. Ultimately, we intend to evaluate the design in a cycle-accurate hardware simulator. This will allow us to comprehensively quantify the impact of design alternatives on cache misses, interrupts and overall performance over several generations of hardware.

This paper is organized as follows. Section 2 provides motivation and background. Section 3 presents our design, and the current prototype implementation is described in Section 4. Section 5 presents our experimental infrastructure and results. Section 6 surveys and contrasts related work, and Section 7 summarizes our contributions and plans for future work.

2 Motivation and Background

To provide the proper motivation and background for our work, we first describe the current best practices of techniques and optimizations for network server performance. Using industry standard benchmarks we then show that, despite these practices, servers are still not scaling with CPU speeds via several benchmarks. Since TCP offload has been a controversial topic in the research community, we review the critiques of offload, providing counterarguments to each point. How TCP offload addresses these scaling issues is described in more detail in Section 3.

2.1 Current Best Practices

Current high-performance servers have adopted many techniques to maximize performance. We provide a brief overview of them here.

Send file with zero copy. Most operating systems have a `sendfile` or `transmitfile` operation that allows sending a file over a socket without copying the contents of the file into user space. This can have substantial performance benefits [30]. However, the benefits are limited to send-side processing; it does not affect receive-side processing. In addition, it requires the server application to maintain its data in the kernel, which may not be feasible

for systems such as application servers, which generate content dynamically.

Checksum offload. Researchers have shown that calculating the IP checksum over the body of the data can be expensive [19]. Most high-performance adapters have the ability to perform the IP checksum over both the contents of the data and the TCP/IP headers. This removes an expensive data-touching operation on both send and receive. However, adapter-level checksums will not catch errors introduced by transferring data over the I/O bus, which has led some to advocate caution with checksum offload [41].

Interrupt coalescing. Researchers have shown that interrupts are costly, and generating an interrupt for each packet arrival can severely throttle a system [28]. In response, adapter vendors have enabled the ability to delay interrupts by a certain amount of time or number of packets in an effort to batch packets per interrupt and amortize the costs [14]. While effective, it can be difficult to determine the proper trigger thresholds for firing interrupts, and large amounts of batching may cause unacceptable latency for an individual connection.

Large send/segmentation offload. TCP/IP implementers have long known that larger MTU sizes provide greater efficiency, both in terms of network utilization (fewer headers per byte transferred) and in terms of host CPU utilization (fewer per-packet operations incurred per byte sent or received). Unfortunately, larger MTU sizes are not usually available due to Ethernet's 1516 byte frame size. Gigabit Ethernet provides "jumbo frames" of 9 KB, but these are only useful in specialized local environments and cannot be preserved across the wide-area Internet. As an approximation, certain operating systems, such as AIX and Linux, provide large send or TCP segmentation offload (TSO) where the TCP/IP stack interacts with the network device as if it had a large MTU size. The device in turn segments the larger buffers into 1516-byte Ethernet frames and adjusts the TCP sequence numbers and checksums accordingly. However, this technique is also limited to send-side processing. In addition, as we demonstrate in Section 2.2, the technique is limited by the way TCP performs congestion control.

Efficient connection management. Early networked servers did not handle large numbers of TCP connections efficiently, for example by using a linear linked-list to manage state [26]. This led to operating systems using hash table based approaches [24] and separating table entries in the `TIME_WAIT` state [2].

Asynchronous interfaces. To maximize concurrency, high-performance servers use asynchronous interfaces as not to block on long-latency operations [33]. Server applications interact using an event notification interface such as `select()` or `poll()`, which in turn can have performance implications [5]. Unfortunately,

Machine	BIOS Release Date	Clock Speed (MHz)	Cycle Time (ns)	Bus Width (bits)	Bus Speed (MHz)	L1 Size (KB)	L2 Size (KB)	E1000 NICs (num)
Workstation-Class								
500 MHz P3	Jul 2000	500	2.000	32	33	32	512	1
933 MHz P3	Mar 2001	933	1.070	32	33	32	256	1
1.7 GHz P4	Sep 2003	1700	0.590	64	66	8	256	2
Server-Class								
450 MHz P2-Xeon	Jan 2000	450	2.200	64	33	32	2048	2
1.6 GHz P4-Xeon	Oct 2001	1600	0.625	64	100	8	256	3
3.2 GHz P4-Xeon	May 2004	3200	0.290	64	133	8	512	4

Table 1: Properties for Multiple Generations of Machines

these interfaces are typically only for network I/O and not file I/O, so they are not as general as they could be.

In-kernel implementations. Context switches, data copies, and system calls can be avoided altogether by implementing the server completely in kernel space [17, 18]. While this provides the best performance, in-kernel implementations are difficult to implement and maintain, and the approach is hard to generalize across multiple applications.

RDMA. Others have also noticed these scaling problems, particularly with respect to data copying, and have offered RDMA as a solution. Interest in RDMA and Infiniband [4] is growing in the local-area case, such as in storage networks or cluster-based supercomputing. However, RDMA requires modifications to both sides of a conversation, whereas Offload can be deployed incrementally on the server side only. Our interest is in supporting existing applications in an inter-operable way, which precludes using RDMA.

While effective, these optimizations are limited in that they do not address the full range of scenarios seen by a server. The main restrictions are: 1) that they do not apply to the receive side, 2) they are not fully asynchronous in the way they interact with the operating system, 3) they do not minimize the interaction with the network interface, or 4) they are not inter-operable. Additionally, many techniques do not address what we believe to be the fundamental performance issue, which is overall server scalability.

2.2 Server Scalability

The recent arrival of 10 gigabit Ethernet and the promise of 40 and 100 gigabit Ethernet in the near future show that raw network bandwidth is scaling at least as quickly as CPU speed. However, it is well-known that memory speeds are not scaling as quickly as CPU speed increases [16]. As a consequence of this and other factors, researchers have observed that the performance of host

TCP/IP implementations is not scaling at the same rate as CPU speeds in spite of raw network bandwidth increases.

To quantify how performance scales over time, we ran a number of experiments using several generations of machines, described in detail in Table 1. We break the machines into 2 classes: desk-side workstations and rack-mounted servers with aggressive memory systems and I/O busses. The workstations include a 500 MHz Intel Pentium 3, a 933 MHz Intel Pentium 3, and a 1.7 GHz Pentium 4. The servers include a 450 MHz Pentium II-Xeon, a 1.6 GHz P4 Xeon, and a 3.2 GHz P4 Xeon. In addition, each of the P4-Xeon servers have 1 MB L3 caches. Each machine runs Linux 2.6.9 and has a number of Intel E1000 MT server gigabit Ethernet adapters, connected via a Dell gigabit switch. Load is generated by five 3.2 GHz P4-Xeons acting as clients, each using an E1000 client gigabit adapter and running Linux 2.6.5. We chose the E1000 MT adapters for the servers since these have been shown to be one of the highest-performing conventional adapters on the market [32], and we did not have access to a 10 gigabit adapter.

We measured the time to access various locations in the memory hierarchy for these machines, including from the L1 and L2 caches, main memory, and the memory-mapped I/O registers on the E1000. Memory hierarchy times were measured using LMBench [25]. To measure the device I/O register times, we added some modifications to the initialization routine of the Linux 2.6.9 E1000 device driver code. Table 2 presents the results. Note that while L1 and L2 access times remain relatively consistent in terms of processor cycles, the time to access main memory and the device registers is increasing over time. If access times were improving at the same rate as CPU speeds, the number of clock cycles would remain constant.

To see how actual server performance is scaling over time, we ran the static portion of SPECweb99 [12] us-

Machine	L1 Cache Hit		L2 Cache Hit		Main Memory		I/O Register Read		I/O Register Write	
	Time (ns)	Clock Cycles	Time (ns)	Clock Cycles	Time (ns)	Clock Cycles	Time (ns)	Clock Cycles	Time (ns)	Clock Cycles
Workstation-Class										
500 MHz P3	6	3	44	22	162	80	600	300	300	150
933 MHz P3	3.25	3	7.5	7	173	161	700	654	400	373
1.7 GHz P4	1.2	2	10.9	18	190	323	800	1355	100	169
Server-Class										
450 MHz P2-Xeon	6.75	3	38.3	17	207	93	800	363	200	90
1.6 GHz Xeon	1.37	2	11.57	18	197	315	900	1440	300	480
3.2 GHz Xeon	0.6	2	5.8	18	111	376	500	1724	200	668

Table 2: Memory Access Times for Multiple Generations of Machines

ing a recent version of Flash [33, 37]. In these experiments, Flash exploits all the available performance optimizations on Linux, including `sendfile()` with zero copy, TSO, and checksum offload on the E1000. Table 3 shows the results. Observe that server performance is not scaling with CPU speed, even though this is a heavily optimized server making use of all current best practices. This is not because of limitations in the network bandwidth; for example, the 3.2 GHz Xeon-based machine has 4 gigabit interfaces and multiple 10 gigabit PCI-X busses.

2.3 Offload: Critiques and Responses

In this paper, we study TCP offload as a solution to the scalability problem. However, TCP offload has been hotly debated by the research community, perhaps best exemplified by Mogul’s paper, “TCP offload is a dumb idea whose time has come” [27]. That paper effectively summarizes the criticisms of TCP offload, and so, we use the structure of that paper to offer our counterarguments here.

Limited processing requirements. One argument is that Clark et al. [9] show that the main issue in TCP performance is implementation, not the TCP protocol itself, and a major factor is data movement; thus Offload does not address the real problem. We point out that Offload does not simply mean TCP header processing; it includes the entire TCP/IP stack, including poorly-scaling, performance-critical components such as data movement, bus crossings, interrupts, and device interaction. Offload provides an improved interface to the adapter that reduces the use of these scalability-limiting operations.

Moore’s Law: Moore’s Law states that CPU speeds are doubling every 18 months, and thus one claim is that Offload cannot compete with general-purpose CPUs. Historically, chips used by adapter vendors have not increased at the same rate as general-purpose CPUs due to

the economies of scale. However, offload can use commodity CPUs with software implementations, which we believe is the proper approach. In addition, speed needs only to be matched with the interface (e.g., 10 gigabit Ethernet), and we argue proper design reduces the code path relative to the non-offloaded case (e.g. with fewer memory copies). Sarkar et al. [38] and Ang [1] show that when the NIC CPU is under-provisioned with respect to the host CPU, performance can actually degrade. Clearly the NIC processing capacity must be sized properly. Finally, increasing CPU speeds does not address the scalability issue, which is what we focus on here.

Efficient host interface: Early critiques are that TCP Offload Engines (TOE) vendors recreated “TCP over a bus”. Development of an elegant and efficient host/adapter interface for offload is a fundamental research challenge, one we are addressing in this paper.

Bad buffer management: Unless Offload engines understand higher-level protocols, there is still an application-layer header copy. While true, copying of application headers is not as performance-critical as copying application data. One complication is the application combining its own headers on the same connection with its data. This can only be solved by changing the application, which is already proposed in RDMA extensions for NFS and iSCSI [7, 8].

Connection management overhead: Unlike conventional NICs, offload adapters must maintain per-connection state. Opponents argue that offload cannot handle large numbers of connections, but Web server workloads have forced host TCP stacks to discover techniques to efficiently manage 10,000’s of connections. These techniques are equally applicable for an interface-based implementation.

Resource management overhead: Critics argue that tracking resource management is “more difficult” for offload. We do not believe this is the case. It is straight-

Machine	Throughput (ops/sec)	Requested Connections	Conforming Connections	Scale (achieved)	Scale (ideal)	Ratio (%)
Workstation-Class						
500 MHz P3	1231	375	375	1.00	1.00	100
933 MHz P3	1318	400	399	1.06	1.87	56
1.7 GHz P4	3457	1200	1169	3.20	3.40	94
Server-Class						
450 MHz P2-Xeon	2230	700	699	1.00	1.00	100
1.6 GHz P4-Xeon	8893	2800	2792	4.00	3.56	112
3.2 GHz P4-Xeon	11614	2500	3490	5.00	7.10	71

Table 3: SPECWeb99 Performance Scalability over Multiple Generations of Machines

forward to extend the notion of resource management across the interface without making the adapter aware of every process as we will show in Sections 3 and 4.

Event management: The claim is that offload does not address managing the large numbers of events that occur in high-volume servers. It is true that offload, per se, does not address *application visible* events, which are better addressed by the API. However, offload can shield *the host operating system* from spurious unnecessary *adapter events*, such as TCP acknowledgments or window advertisements. In addition, it allows batching of other events to amortize the cost of interrupts and bus crossings.

Partial offload is sufficiently effective: Partial offload approaches include checksum offload and large send (or TCP Segmentation Offload), as discussed in Section 2.1. While useful, they have limited value and do not fully solve the scalability problem as was shown in Section 2.2. Other arguments include that checksum offload actually masks errors to the host [41]. In contrast, offload allows larger batching and the opportunity to perform more rigorous error checking (by including the CRC in the data descriptors).

Maintainability: Opponents argue that offload-based approaches are more difficult to update and maintain in the presence of security and bug patches. While this is true of an ASIC-based approach, it is not true of a software-based approach using general-purpose hardware.

Quality assurance: The argument here is that offload is harder to test to determine bugs. However, testing tools such as TBIT [31] and ANVL [11] allow remote testing of the offload interface. In addition, software based approaches based on open-source TCP implementations such as Linux or FreeBSD facilitate both maintainability and quality assurance.

System management interface: Opponents claim that offload adapters cannot have the same management interface as the host OS. This is incorrect: one example

is SNMP. It is trivial to extend this to an offload adapter.

Concerns about NIC vendors: Third-party vendors may go out of business and strand the customer. This has nothing to do with offload; it is true of any I/O device: disk, NIC, or graphics card. Economic incentives seem to address customer needs. In addition, one of the largest NIC vendors is Intel.

3 System Design

In this Section we describe our Offload design and how it addresses scalability.

3.1 How Offload Addresses Scalability

A higher-level interface. Offload allows the host operating system to interact with the device at a higher level of abstraction. Rather than simply queuing MTU-sized packets for transmission or reception, the host issues commands at the transport layer (e.g., `connect()`, `accept()`, `send()`, `close()`). This allows the adapter to shield the host from transport layer events (and their attendant interrupt costs) that may be of no interest to the host, such as arrivals of TCP acknowledgments or window updates. Instead, the host is only notified of meaningful events. Examples include a completed connection establishment or termination (rather than every packet arrival for the 3-way handshake or 4-way tear-down) or application-level data units. Sufficient intelligence on the adapter can determine the appropriate time to transfer data to the host, either through knowledge of standardized higher-level protocols (such as HTTP or NFS) or through a programmable interface that can provide an application signature (i.e., an application-level equivalent to a packet filter). By interacting at this higher level of abstraction, the host will transfer less data over the bus and incur fewer interrupts and device register accesses.

Ability to move data in larger sizes. As described in Section 2.1, the ability to use large MTUs has a significant impact on performance for both sending and re-

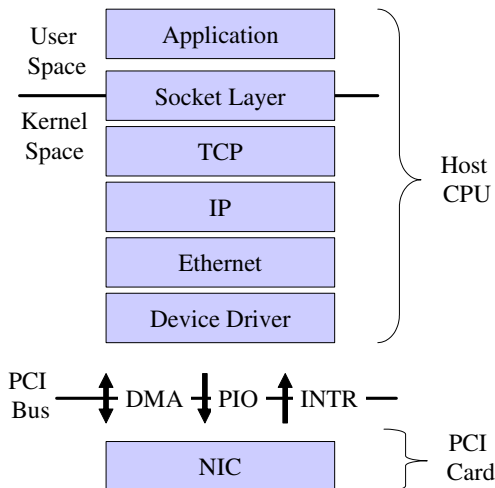


Figure 1: Conventional Protocol Stack

ceiving data. Large send/TSO only approximates this optimization, and only for the send side. In contrast, ofoad allows the host to send and receive data in large chunks unaffected by the underlying MTU size. This reduces use of poorly scaling components by making more efficient use of the I/O bus. Utilization of the I/O bus is not only affected by the data sent over it, but also by the DMA descriptors required to describe that data; ofoad reduces both. In addition, data that is typically DMA'ed over the I/O bus in the conventional case is not transferred here, for example TCP/IP and Ethernet headers.

Improving memory reference behavior. We believe ofoad will not only increase available cycles to the application but improve application memory reference behavior. By reducing cache and TLB pollution, cache hit rates and CPI will improve, increasing application performance.

3.2 Current Adapter Designs

Perhaps the simplest way to understand an architecture that ofoads all TCP/IP processing is to outline the ways in which ofoad differs from conventional adapters in the way it interacts with the OS. Figure 1 illustrates a conventional protocol architecture in an operating system. Operating systems tend to communicate with conventional adapters only in terms of data transfer by providing them with two queues of buffers. One queue is made up of ready-made packets for transmission; the other is a queue of empty buffers to use for packet reception. Each queue of buffers is identified, in turn, by a descriptor table that describes the size and location of each buffer in the queue. Buffers are typically described in physical memory and must be pinned to ensure that they

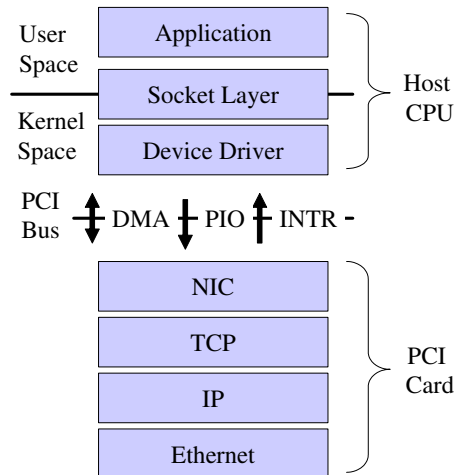


Figure 2: Offload Architecture

are accessible to the card, i.e., so that they are not paged out. The adapter provides a memory-mapped I/O interface for telling the adapter where the descriptor tables are located in physical memory, and provides an interface for some control information, such as what interrupt number to raise when a packet arrives. Communication between the host CPU and the adapter tends to be in one of three forms, as is shown in Figure 1: DMA's of buffers and descriptors to and from the adapter; reads and writes of control information to and from the adapter, and interrupts generated by the adapter.

3.3 Offloaded Adapter Design

An architecture that seeks to ofoad the full TCP/IP stack has both similarities and differences in the way it interacts with the adapter. Figure 2 illustrates our ofoad architecture. As in the conventional scenario, queues of buffers and descriptor tables are passed between the host CPU and the adapter, and DMA's, reads, writes and interrupts are used to communicate. In the ofoad architecture, however, the host and the adapter communicate using a higher level of abstraction. Buffers have more explicit data structures imposed on them that indicate both control and data interfaces. As with a conventional adapter, passed buffers must be expressed as physical addresses and must be in pinned memory. The control interface allows for the host to command the adapter (e.g., what port numbers to listen on) and for the adapter to instruct the host (e.g., to notify the host of the arrival of a new connection). The control interface is invoked, for example, by conventional socket functions that control connections: `socket()`, `bind()`, `listen()`, `connect()`, `accept()`, `setsock-`

`opt()`, etc. The data interface provides a way to transfer data on established connections for both sending and receiving and is invoked by socket functions such as `send()`, `sendto()`, `write()`, `writenv()`, `read()`, `readv()`, etc. Even the data interface is at a higher layer of abstraction, since the passed buffers consist of application-specific data rather than fully-formed Ethernet frames with TCP/IP headers attached. In addition, these buffers need to identify which connection that the data is for. Buffers containing data can be in units much larger than the packet MTU size. While conceptually they could be of any size, in practice they are unlikely to be larger than a VM page size.

As with a conventional adapter, the interface to the offload adapter need not be synchronous. The host OS can queue requests to the adapter, continue doing other processing, and then receive a notification (perhaps in the form of an interrupt) that the operation is complete. The host can implement synchronous socket operations by using the asynchronous interface and then block the application until the results are returned from the adapter. We believe asynchronous operation is key in order to ameliorate and amortize fixed overheads. Asynchrony allows larger-scale batching and enables other optimizations such as polling-based approaches to servers [3, 28].

The offload interface allows supporting conventional user-level APIs, such as the socket interface, as well as newer APIs that allow more direct access to user memory such as DAFS, SDP, and RDMA. In addition, offload allows performing *zero-copy* sends and receives *without changes to the socket API*. The term zero-copy refers to the elimination of memory-to-memory copies by the host. Even in the zero-copy case, data is still transferred across the I/O bus by the adapter via DMA.

For example, in the case of a send using a conventional adapter, the host typically copies the data from user space into to a pinned kernel buffer, which is then queued to the adapter for transmission. With an intelligent adapter, the host can block the user application and pin its buffers, then invoke the adapter to DMA the data directly from the user application buffer. This is similar to previous “single-copy” approaches [13, 20], except that the transfer across the bus is done by the adapter DMA and not via an explicit copy by the host CPU.

Observe from Figure 2 that the interaction between the host and the adapter now occurs between the socket and TCP layers. A naive implementation may make unnecessary transfers across the PCI bus for achieving socket functionality. For example, `accept()` would now cause a bus crossing in addition to a kernel crossing, as could `setsockopt()` for actions such as changing the send or receive buffer sizes or the Nagle algorithm. However, each of these costs can be amortized via batching multiple requests into a single request that crosses the bus.

For example, multiple arrived connections can be aggregated into a single `accept()` crossing which then translates into multiple `accept()` system calls. On the other hand, certain events that would generate bus crossings with a conventional adapter might not do so with a offload adapter, such as ACK processing and generation. The relative weight of these advantages and disadvantages depends on the implementation and workload of the application using the adapter.

4 System Implementation

To evaluate our design and the impact of design decisions, we implemented a software prototype. Our decision to implement the prototype purely in software, rather than building or modifying actual adapter hardware, was motivated by several factors. Since our goal is to study not just performance, but scalability, we ultimately intend to model different hardware characteristics, for both the host and adapter, using a cycle accurate hardware simulator. Limiting our analysis to only currently available hardware would hinder our evaluation for future hardware generations. Ultimately, we envision an adapter with a general purpose processor, in addition to specialized hardware to accelerate specific operations such as checksum calculation. Our prototype software is intended to serve as a reference implementation for a production adapter.

Our prototype is composed of three main components:

- OSLayer, an operating system layer that provides the socket interface to applications and maps it to the descriptor interface shared with the adapter;
- Event-driven TCP, our offloaded TCP implementation;
- IOLib, a library that encapsulates interaction between OSLayer and Event-driven TCP.

At the moment, OSLayer is implemented as a library that is statically linked with the application. Ultimately, it will be decomposed into two components: a library linked with applications and a component built in the kernel. Event-driven TCP currently runs as a user-level process that accesses the actual network via a raw socket. It will eventually become the main software loop on the adapter. The IOLib implementation currently communicates via TCP sockets, but the design allows for implementations that communicate over a PCI bus or other interconnects such as Infiniband. This provides a vehicle for experimentation and analysis and allows us to measure bus traffic without having to build a detailed simulation of a PCI bus or other interconnect.

We used the Flash Web server for our evaluation, with Flash and OSLayer running on one machine and Event-driven TCP running on another. We use `httperf` [29] running on a separate machine to drive load. To compare

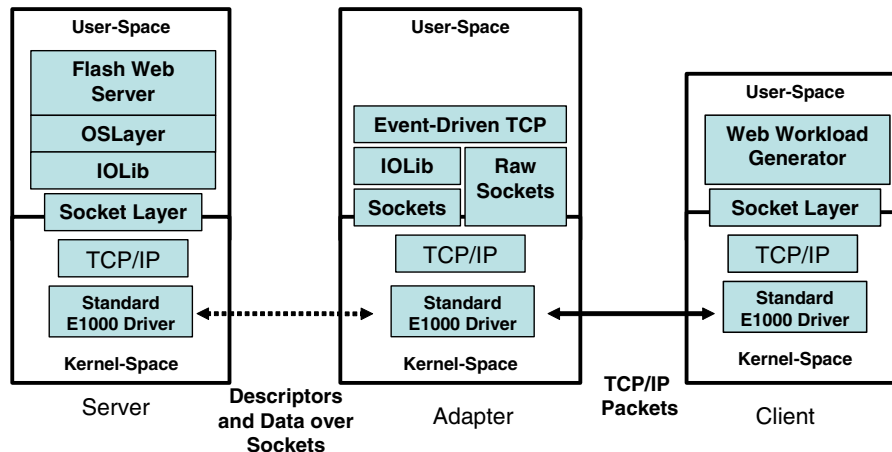


Figure 3: Offload Prototype

the behavior of the prototype with the conventional case, we evaluate a similar client-server configuration using an E1000 device driver that has been instrumented to measure bus traffic. Figure 3 illustrates how the components fit together. The implementation is described in more detail below.

4.1 OSLayer

OSLayer is essentially the socket interface decoupled from the TCP protocol implementation. OSLayer is a library that exposes an asynchronous socket interface to network applications. As seen in Figure 3, the application employs the socket API and OSLayer communicates with Event-driven TCP via IOLib through descriptors, discussed in more detail in Section 4.3. After creating the descriptor appropriate to the particular API function, the call is returned.

OSLayer and Event-driven TCP interact via a byte stream abstraction. Towards that end, OSLayer transfers buffers of 4 KB to Event-driven TCP. For large transfers, this reduces the number of DMA's and bus crossings significantly.

To further limit bus crossings and increase scalability, OSLayer employs several techniques. Descriptors can be batched before transferring them to the card. In the current implementation, we allow batching using a configurable batching level. However, a timer is used to ship over descriptors that have been waiting sufficiently long before the batching level has been reached. Even if batching is set to one, descriptors can still batch significantly with large data transfers. In addition, OSLayer performs buffer coalescing (similar to the TCP_CORK socket option on Linux), which is utilized by the Flash Web server. When using the `sendfile()` operation, this allows HTTP headers and data to be aggregated,

thus sharing a descriptor and therefore a transfer. While the conventional Linux stack is limited to two `sk_buffs`, OSLayer can combine any number of `sk_buffs` into one.

4.2 Event Driven TCP

Event-driven TCP (EDT) performs the majority of the TCP processing for the adapter and was derived from the Arsenic user-level TCP stack [34]. Normally a TCP stack running on the host is animated by three types of events: a calling user-space application process or thread, a packet arrival, or a timer interrupt. Since there are no applications on the adapter, an event-driven architecture was chosen since it scales better than a process or thread-based approach. EDT is thus a single-threaded event-based closed loop, implemented as a stand-alone user-space process. On every iteration of the loop, each of the following are checked: pending packets, new descriptors, DMA completion and TCP timers. Execution is thus animated by packet and descriptor arrivals, DMA completions, and TCP timer firings.

Event-driven TCP does not necessarily notify OSLayer of every packet. For example, instead of informing OSLayer about every acknowledgment, OSLayer is only alerted when an entire transfer completes. OSLayer receives only a single event for a connection establishment or termination, rather than each packet of the TCP handshake. This reduces the number of descriptors (and their corresponding events) to be transferred and processed by the host.

EDT communicates to OSLayer by passing descriptors through IOLib, discussed in Sections 4.3 and 4.4. Since it is a user-space process, EDT sends and receives packets over the network using raw sockets and `libpcap` [21].

4.3 Descriptors

Descriptors are a software abstraction intended to capture the hardware-level communication mechanism that occurs over the I/O bus between host and adapter. Descriptors are primarily typed by how they are used: *request* descriptors for issuing commands (e.g., CONNECT, SEND), and *response* descriptors for the results of those commands (e.g., success, failure, retry, etc.). Descriptors are further categorized as control (SOCKET, BIND, etc.) or data (SEND, RECV). Two separate sets of tables are used for each transfer direction.

When an application calls send, a SEND request descriptor is transferred from OSLayer to Event-driven TCP, containing the address and length of the buffer to be sent. A request for DMA is queued at Event-driven TCP and the next descriptor is processed. After the DMA completes, the event is picked up by Event-driven TCP, and a response descriptor is created with the address of the buffer. This descriptor informs OSLayer that the buffer is no longer used. Upon receipt of this SEND response, OSLayer cleans up the send buffer. Of course, many send descriptors can be sent to Event-driven TCP at once. Buffers described by a SEND descriptor can be up to 4 KB. We chose 4 KB since this is the standard page size in most architectures; however, our implementation has the ability to transfer up to 64K bytes.

When receive() is called, the RECV descriptor is transferred from OSLayer to Event-driven TCP, containing the address and length of the buffer to DMA received data into. If data is available on Event-driven TCP's receive queue, a DMA is immediately initiated. Later, after DMA is finished, a RECV response descriptor is created, notifying OSLayer that the data is available, and Event-driven TCP can free its own sk_buff. If data is not available upon receipt of a RECV descriptor, the buffer is placed on a receive buffer queue for that connection. When the data does arrive later on the appropriate connection, the buffer is removed from the queue, the DMA is performed, and a RECV response descriptor is created and sent to the host.

Most of the control descriptors work in a relatively straightforward manner; however, the CLOSE operation is worth describing in more detail. A CLOSE descriptor is transferred from OSLayer to Event-driven TCP when the application initiates a close. After sending out all of the sk_buffs on the write queue, Event-driven TCP will signal the close to the remote peer via sending a packet with the FIN bit set. After the final ACK is sent a response descriptor is created. In the event the other side closes the connection, a CLOSE command descriptor is created in Event-driven TCP and sent to OSLayer. OSLayer need not reply with a CLOSE response descriptor in this case; OSLayer just notifies the application and cleans up appropriately.

All DMA's involving data are initiated by Event-driven TCP, allowing EDT to control the flow up to the host. Note that a DMA is not necessarily performed immediately. Since the request for a DMA is queued, it may be some time before a response to a descriptor is received by OSLayer.

This queued DMA approach required some changes to the TCP stack because pending sends were not preventing a CLOSE descriptor from being processed before the send's DMA competed. Since it was difficult to determine how many sends were queued for DMA (and when they were finished), "empty" sk_buffs are placed on the write queue with a flag set indicating that the data is not yet present. When the DMA completes, this flag is set to true, and the sk_buff is ready for sending. Thus, this flag is checked before sending any sk_buff. This caused changes in several components in the TCP stack. For example, close processing is now split into two pieces. The first part indicates the connection is in the process of closing; The second part actually completes the close, after the last DMA is complete and the buffer is sent.

4.4 IOLib

IOLib provides a communication library to the OSLayer and Event-driven TCP code by abstracting the I/O layer to a generic Put/Get interface. We chose this approach for ease of porting the offload prototype to bus, fabric or serial communication interfaces. Thus, only IOLib needs to understand the specific properties of the underlying communication link, while the calls within OSLayer and Event-driven TCP remain unchanged.

The IOLib Put/Get library has an asynchronous queuing interface for sending and receiving data. This interface is augmented by virtual interface registers that can be used for base address references traditionally used in the PCI bus interface. Communications support for the Put/Get interface can be provided by several types of communication: shared memory, message passing, etc. Figure 3 shows an example of how the server and adapter components communicate using IOLib, where support for the Put/Get interface is provided over a standard TCP/IP socket.

Since IOLib provides the interface between the host and the adapter, it is a natural place to monitor traffic between the two. To facilitate comparisons to conventional adapter implementations, we instrumented IOLib to measure three different aspects of I/O traffic: number of DMA's requested, number of bytes transferred, and number of I/O bus cycles consumed by a transfer. The model for capturing the number of bus cycles consumed is based on a 133 MHz, 64 bit PCI-X bus and is calculated as follows:

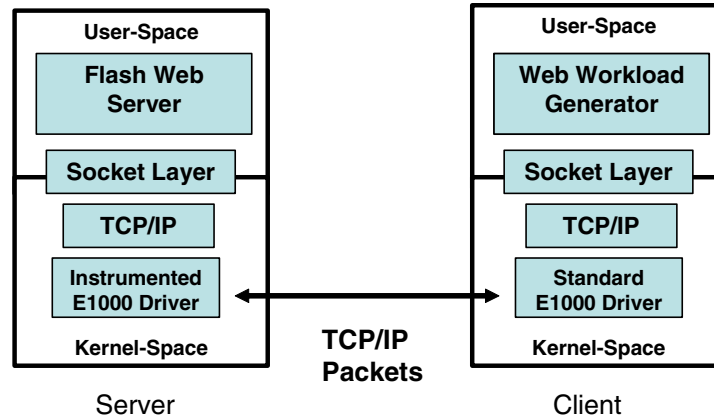


Figure 4: Baseline Configuration

$$cycles = 4 + ((transfer_size + 7)/8)$$

This is because four cycles are required for initiation and termination [10], the bus is eight bytes (64 bits) wide, and transfers that are less than a full multiple of eight consume the bus for the entire cycle. Charges are incurred for all data transferred; not only the packet buffers transferred but also for the DMA descriptors that describe them.

4.5 Limitations of the Prototype

OSLayer is still under active development. Many of the socket options not used by Flash are not fully implemented. OSLayer requires a single-threaded application because there is no current mechanism to distinguish descriptors between threads or processes. A feedback mechanism is still required so that OSLayer knows how many send buffers are available in Event-driven TCP. If there are no send buffers available, then OSLayer can return a failure code to the application invoking the send.

More work can be done to improve and extend Event-driven TCP. For example, it could be made current with the latest version of the Linux TCP stack. We believe performance would be improved if `sk_buffs` could reference multiple noncontiguous pages.

Certain non-essential descriptors are not yet implemented. An immediate mode descriptor, that is, one with the data embedded directly in the descriptor, would also reduce the number of bus crossings. Descriptors for sending status (e.g., the number of available send buffers) and option querying could also improve performance and allow more dynamic behavior. Finally, descriptors to cancel a send or a receive are needed.

Several new operations with associated descriptors are planned. A batching `ACCEPT` operation will allow OSLayer to instruct Event-driven TCP to wait for *N* connections to be established before returning a response to the host. A single response descriptor would contain all of the requisite information about each connection. In the appropriate scenarios, this should reduce `ACCEPT` descriptor traffic. The next logical step is to provide the option of delaying the connection notification until the arrival of the first data on a newly-established connection. Another item is the addition of a “close” option to a `SEND` descriptor that lets a close operation be combined with a send. This eliminates the need for a separate close descriptor, and can increase the likelihood that the `FIN` bit is piggybacked on the final data segment.

We are also designing cumulative completion descriptors. Instead of completing each send or receive request individually with its own `SEND/RECV` complete descriptor, we intend to have a send complete descriptor that indicates completion of all requests up to and including that one. This change requires no syntactic changes to the descriptors; it simply changes the semantics of the response so that completion of a send/receive implicitly indicates completion of any previous sends. This approach is employed by OE [42], and we believe the benefits can be achieved in our stack as well.

4.6 E1000 Driver

To provide comparisons with a baseline system, we modified the Linux 2.6.9 Intel E1000 device driver code to measure the same three components of bus traffic as was done for IOLib: DMA’s requested, bytes transferred, and bus cycles consumed. The bus model is the same as is described in Section 4.4. Sends are measured in `e1000_tx_queue()`; receives are monitored

	1 KB			64 KB			512 KB		
	E1000 (num)	Offload (num)	Diff (%)	E1000 (num)	Offload (num)	Diff (%)	E1000 (num)	Offload (num)	Diff (%)
Recv DMA Count	12	11	08	62	39	37	363	189	48
Recv Bus Cycles Consumed	119	78	34	559	269	52	3260	1394	57
Recv Bytes Transferred	538	252	53	2385	821	66	13900	4706	66
Send DMA Count	9	9	0	159	56	64	1222	370	70
Send Bus Cycles Consumed	237	200	16	9366	8532	9	74244	67683	8
Send Bytes Transferred	1572	1301	17	69474	66389	4	552132	529131	4

Table 4: Comparing IO Traffic for E1000 and Offload

in `e1000_clean_rx_irq()`. Figure 4 shows how the instrumented driver is used in our experiments.

5 Experimental Results

In this Section, we present the results of our prototype described in Section 4 and compare it to a baseline implementation meant to represent the current state of the art in conventional (i.e., non-offloaded) systems. The goal is to show that offload provides a more efficient interface between the host and the adapter for the metric we are able to measure, namely, I/O traffic. We again use a simple Web server workload to evaluate our prototype. For software we use the Flash Web server [33] and the `httperf` [29] client workload generator. We use multiple nodes within an IBM Blade Center to produce the offload prototype configuration depicted in Figure 3. The baseline configuration is shown in Figure 4.

We examine three scenarios: transferring a small file (1 KB), a moderately-sized file (64 KB), and a large file (512 KB). This is intended to capture a spectrum of data transfer sizes and vary the ratio of per-connection costs to per-byte costs. We measure transfers in both directions (send and receive) using three metrics for utilizing the I/O bus: *DMA count*, which counts the number of times a DMA is requested from the bus; *bus cycles*, which measures the number of cycles consumed on the bus (based on the model in Section 4.4); and *bytes transferred*, to determine the raw number of bytes sent over the bus.

5.1 Baseline Results

Table 4 shows our results. Overall, we see that offload is effective at reducing bus activity, with improvements up to 70 percent. We look at each transfer size in turn. Examining the results for 1 KB transfers in Table 4, note that there is a significant improvement on the receive path, mainly due to shielding the host from ACK packets. However, this is a send-side test, and the number of bytes sent from application to application do not change. Even so, we see a moderate reduction (4-17 %) in bytes

transferred on the send side. This is partly because TCP, IP and Ethernet headers are not transferred over the bus in the offload prototype, whereas they are in the baseline case. Note that the number of DMAs and the utilization of the bus are also reduced, up to 70 % and 16 %, respectively.

Looking at the results for 64 KB transfers, again we see significant improvement on the receive side. A larger amount of data is being sent in this experiment, and thus the byte savings on the send side are relatively small at four percent. However, note that the efficiency of the bus has greatly improved: the number of send DMA's requested falls by 64 percent, and the bus utilization is reduced by 9 percent. The amount of bus cycles consumed has also improved by 9 percent. These trends are also reflected in the 512 KB results.

5.2 Batching Descriptors

One obvious method to reduce bus crossings is to transfer multiple descriptors at a time rather than one. The results presented in Table 5 provide experimental results for a minimum batching level of ten descriptors at a time, using a idle timeout value of ten milliseconds. These can be tuned to the transfer size, but are held constant for these experiments.

Observe that the numbers have improved for the 1K, 64K and 512K transfers over the previous comparison in Table 4. The improvements are limited since increasing the minimum batching threshold and timeouts did not significantly help for this type of traffic. This is because multiple response and socket descriptor messages are provided at nearly the same time. This technique is similar in concept to interrupt coalescing in adapters; the distinction is that information batched at a higher level of abstraction.

6 Related Work

Several performance studies on TCP offload have been conducted using an emulation approach which partitions an SMP and uses a processor as an offload engine. These

	1 KB			64 KB			512 KB		
	E1000 (num)	Offload (num)	Diff (%)	E1000 (num)	Offload (num)	Diff (%)	E1000 (num)	Offload (num)	Diff (%)
Recv DMA Count	12	7	42	62	9	85	363	24	93
Recv Bus Cycles Consumed	119	60	50	559	134	76	3260	648	80
Recv Bytes Transferred	538	244	55	2385	761	68	13900	4375	69
Send DMA Count	9	5	44	159	21	87	1222	148	88
Send Bus Cycles Consumed	237	183	23	9366	8385	11	74244	66683	10
Send Bytes Transferred	1572	1294	18	69474	66319	5	552132	528686	4

Table 5: Comparing IO Traffic for E1000 and Offload Batching Descriptor Traffic.

studies have shown that offloading TCP processing to an intelligent interface can provide significant performance improvements when compared to the standard TCP/IP networking stack. However, the study by Westrelin et al. [42] lacks an effective way to model the I/O bus traffic that occurs between the host and offload adapter. They use the host memory bus to emulate the I/O bus, but this emulation lacks the characteristics necessary to capture the performance impact of an I/O bus such as PCI. In practice, a high-speed memory bus is not representative of the performance seen by an I/O bus. Our implementation is designed with a modular I/O library that can be used to model different I/O bus types. The focus of our paper considers multiple performance impacts on server scalability including I/O. Additionally, when using a partitioned SMP emulation approach, there is coherency traffic necessary to keep the memory state consistent between processors. This coherency overhead can affect the results, since it perturbs the interaction between the host and offload adapter and includes overhead that will not exist in a real system. Our modeled offload system does not suffer from this issue.

Rangarajan et al. [35], and Regnier et al. [36] also use a partitioned SMP approach and show greater absolute performance when dedicating a processor to packet processing. This approach can measure server scalability with respect to the CPU but does not address the underlying scalability issues that exist in other parts of the system, such as the memory bus.

TCP offload designs that do not address the scalability issues discussed in this paper might improve CPU utilization on the host for large block sizes but harm throughput and latency for small block sizes. The current generation of offload adapters in the market have simply moved the TCP stack from the host to the offload adapter without the necessary design considerations for the host and adapter interface. For some workloads this creates a bottleneck on the adapter [38]. Handshaking across the host and adapter interface can be quite costly and reduce performance especially for small messages.

Additionally, Ang [1] found that there appears to be no cheap way of moving data between host memory and an intelligent interface.

Performance analysis of current generation network adapters only reveals the characteristics of networking at a given point in time. In order to understand the performance impacts of various design tradeoffs, all of the components of the system need to be modeled so that performance characteristics that change over time can be revealed. Binkert et al. [6] propose the execution-driven simulator M5 to model network-intensive workloads. M5 is capable of full system simulation including the OS, the memory model, caching effects, DMA activity and multiple networked systems. M5 faithfully models the system so it can boot an unmodified OS kernel and execute applications in the simulated environment. In Section 7 we describe the use of Mambo, an instruction level simulator for the PowerPC®, in order to faithfully model network-intensive workloads.

Shivam and Chase [40] showed that offload can enable direct data placement, which can serve to eliminate some communication overheads, rather than of shifting them from the host to the adapter. They also provide a simple model to quantify the benefits of offload based on the ratio of communication to computation and the ratio of the host CPU processing power to the NIC processing power. Thus a workload can be characterized based on the parameters of the model and one can determine whether offload will benefit that workload. This paper can be seen as an application of Amdahl's Law to TCP offload. Their analysis suggests that offload best supports low-lambda applications such as storage servers.

Foong et al. [15] found that performance is scaling at about 60 percent of CPU speeds. This implies that generally accepted rule of thumb that states 1 bps of network link requires 1 Hz of CPU processing will not hold up over time. They point out that as CPU speed increases the performance gap widens between it and the memory and I/O bus. However, their study did not generate an implementation and their results are from using an em-

ulated offload system. Our work has focused on these server scalability issues and created a design and implementation to study them.

7 Summary and Future Work

We have presented experimental evidence that quantifies how poorly server network throughput is scaling with CPU speed even with sufficient link and I/O bandwidth. We argue the scalability problem is due to specific operations that limit scalability, in particular bus crossings, cache misses and interrupts. Furthermore, we have shown experimental evidence that quantifies how bus crossings and cache misses are scaling poorly with CPU speed. We have designed a new host/adaptor interface that exploits additional processing at the network interface to reduce scalability-limiting operations. Experiments with a software prototype of our offloaded TCP stack show that it can substantially reduce bus crossings. By allowing the host to deal with network data in fewer pieces, we expect our design to reduce cache misses and interrupts as well. Work is ongoing to continue development of the prototype and extend our analysis to study the effects on cache misses and interrupts.

As described in Section 4.5, the current prototype does not yet implement all aspects of the design. We are continuing development with an emphasis on further aggregation and reduction of operations that limit scalability. Future additions include a batching accept operation, an accept that returns after data arrives on the connection, a send-and-close function, and cumulative completion semantics.

We are also preparing to evaluate our prototype in Mambo, a simulation environment for PowerPC[®] systems [39]. Running in Mambo provides the ability to measure cache behavior and quantify the impact of hardware parameters such as processor clock rates, cache sizes, associativity and miss penalties. Mambo allows us to run the OSLayer (host) and Event-driven TCP (adaptor) portions of the prototype on distinct simulated processors. We can thus determine the hardware resources needed on the adaptor to support a given host workload.

Finally, we intend to extend the prototype and simulation to encompass low-level device interaction. This will entail replacing the socket-based version of IOLib with a version that communicates across a hardware interconnect such as PCI or InfiniBand[®]. This will allow us to predict throughput and latency on simulated next-generation interconnects.

References

- [1] Boon S. Ang. An evaluation of an attempt at offloading TCP/IP protocol processing onto an i960rn-based NIC.

Technical Report 2001-8, HP Labs, Palo Alto, CA, Jan 2001.

- [2] Mohit Aron and Peter Druschel. TCP implementation enhancements for improving Webserver performance. Technical Report TR99-335, Rice University Computer Science Dept., July 1999.
- [3] Mohit Aron and Peter Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.
- [4] The Infiniband Trade Association. The Infiniband architecture. <http://www.infinibandta.org/specs>.
- [5] Gaurav Banga, Jeffrey Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX 1999 Technical Conference*, Monterey, CA, June 1999.
- [6] Nathan L. Binkert, Erik G. Hallnor, and Steven Reinhardt. Network-oriented full-system simulation with M5. In *Proceedings Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads CAECW*, Anaheim, CA, Feb 2003.
- [7] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. NFS over RDMA. In *Proceedings ACM SigComm Workshop on Network-I/O Convergence (NICELI)*, Karlsruhe, Germany, Aug 2003.
- [8] Mallikarjun Chadalapaka, Uri Elzur, Michael Ko, Hemal Shah, and Patricia Thaler. A study of iSCSI extensions for RDMA (iSER). In *Proceedings ACM SigComm Workshop on Network-I/O Convergence (NICELI)*, Karlsruhe, Germany, Aug 2003.
- [9] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6), June 1989.
- [10] Intel Corporation. Small packet traffic performance optimization for 8255x and 8254x Ethernet controllers. Technical Report Application Note (AP-453), Sept 2003.
- [11] Ixia Corporation. ANVL TCP testing tool. <http://www.ixiacom.com/>.
- [12] The Standard Performance Evaluation Corporation. SPECWeb99. <http://www.spec.org/osg/web99, 1999>.
- [13] Chris Dalton, Greg Watson, David Banks, Costas Clamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, 11(2):36–43, July 1993.
- [14] Peter Druschel, Larry Peterson, and Bruce Davie. Experiences with a high-speed network adaptor: A software perspective. In *ACM SIGCOMM Symposium*, London, England, August 1994.
- [15] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, and Greg J. Regnier. TCP performance re-visited. In *Proceedings International Symposium on Performance Analysis of Systems and Software ISPASS*, Austin, TX, March 2003.

- [16] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (2nd Edition)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1995.
- [17] Red Hat Inc. The Tux WWW server. <http://people.redhat.com/~mingo/TUX-patches/>.
- [18] Philippe Joubert, Robert King, Richard Neves, Mark Russinovich, and John Tracey. High-performance memory-based Web servers: Kernel and user-space performance. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.
- [19] Jonathan Kay and Joseph Pasquale. Profiling and reducing processing overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, 4(6):817–828, December 1996.
- [20] Karl Kleinpaste, Peter Steenkiste, and Brian Zill. Software support for outboard buffering and checksumming. In *ACM SIGCOMM Symposium*, pages 196–205, Cambridge, MA, August 1995.
- [21] The libpcap Project. <http://sourceforge.net/projects/libpcap/>.
- [22] Srihari Makineni and Ravi Iyer. Measurement-based analysis of TCP/IP processing requirements. In *10th International Conference on High Performance Computing (HiPC 2003)*, Hyderabad, India, December 2003.
- [23] Evangelos P. Markatos. Speeding up TCP/IP: Faster processors are not enough. In *Proceedings 21st IEEE International Performance, Computing, and Communication Conference IPCCC*, pages 341–345, Phoenix, AZ, April 2002.
- [24] Paul E. McKenney and Ken F. Dove. Efficient demultiplexing of incoming TCP packets. In *ACM SIGCOMM Symposium*, pages 269–279, Baltimore, Maryland, August 1992. ACM.
- [25] Larry McVoy and Carl Staelin. LMBENCH: Portable tools for performance analysis. In *USENIX Technical Conference of UNIX and Advanced Computing Systems*, San Diego, CA, January 1996.
- [26] Jeffrey C. Mogul. Operating systems support for busy Internet servers. In *Proceedings Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.
- [27] Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *USENIX Workshop on Hot Topics on Operating Systems (HotOS)*, Hawaii, May 2003.
- [28] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.
- [29] David Mosberger and Tai Jin. httpperf – a tool for measuring Web server performance. In *Proceedings 1998 Workshop on Internet Server Performance (WISP)*, Madison, WI, June 1998.
- [30] Erich M. Nahum, Tsipora Barzilai, and Dilip Kandlur. Performance issues in WWW servers. *IEEE/ACM Transactions on Networking*, 10(2):2–11, Feb 2002.
- [31] Jitendra Padhye and Sally Floyd. On inferring TCP behavior. In *ACM SIGCOMM Symposium*, pages 287–298, 2001.
- [32] Vijay Pai, Scott Rixner, and Hyong-Youb Kim. Isolating the performance impacts of network interface cards through microbenchmarks. In *Proceedings ACM Sigmetrics*, New York, NY, June 2004.
- [33] Vivek Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference*, Monterey, CA, June 1999.
- [34] Ian Pratt and Keir Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Anchorage, Alaska, April 2001.
- [35] Murali Rangarajan, Aniruddha Bohra, Kalpana Banerjee, Enrique V. Carrera, Ricardo Bianchini, and Liviu Iftode. TCP servers: Offloading TCP processing in Internet servers, design, implementation and performance. Technical Report DCS-TR-481, Rutgers University, Department of Computer Science, Piscataway, NJ, March 2003.
- [36] Greg Regnier, Dave Minturn, Gary McAlpine, Vikram Saletore, and Annie Foong. ETA: Experience with an intel xeon processor as a packet processing engine. In *11th Annual Symposium on High Performance Interconnects*, Palo Alto, CA, August 2003.
- [37] Yaoping Ruan and Vivek Pai. Making the “box” transparent: System call performance as a first-class result. In *USENIX Annual Technical Conference*, Boston, MA, June 2004.
- [38] Prasenjit Sarkar, Sandeep Uttamchandani, and Kaladhar Voruganti. Storage over IP: When does hardware support help? In *USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, March 2003.
- [39] H. Shafi, P.J. Bohrer, J. Phelan, C.A. Rusu, and J.L. Peterson. Design and validation of a performance and power simulator for POWERPC systems. *IBM Journal of Research and Development*, 47(5/6):641–651, September/November 2003.
- [40] Piyush Shivam and Jeffrey S. Chase. On the elusive benefits of protocol offload. In *ACM SigComm Workshop on Network-IO Convergence (NICELI)*, Germany, August 2003.
- [41] Jonathan Stone, Michael Greenwald, Craig Partridge, and James Hughes. Performance of checksums and CRC’s over real data. *IEEE/ACM Transactions on Networking*, 6(5):529–543, 1998.
- [42] R. Westrelin, N. Fugier, E. Nordmark, K. Kunze, and E. Lemoine. Studying network protocol offload with emulation: Approach and preliminary results. In *12th Annual IEEE Symposium on High Performance Interconnects*, Stanford, CA, Aug 2004.

A Portable Kernel Abstraction For Low-Overhead Ephemeral Mapping Management

Khaled Elmeleegy, Anupam Chanda, and Alan L. Cox

Department of Computer Science

Rice University, Houston, Texas 77005, USA

{kdiaa, anupamc, alc}@cs.rice.edu

Willy Zwaenepoel

School of Computer and Communication Sciences

EPFL, Lausanne, Switzerland

willy.zwaenepoel@epfl.ch

Abstract

Modern operating systems create ephemeral virtual-to-physical mappings for a variety of purposes, ranging from the implementation of inter-process communication to the implementation of process tracing and debugging. With succeeding generations of processors the cost of creating ephemeral mappings is increasing, particularly when an ephemeral mapping is shared by multiple processors.

To reduce the cost of ephemeral mapping management within an operating system kernel, we introduce the `sf_buf` ephemeral mapping interface. We demonstrate how in several kernel subsystems — including pipes, memory disks, sockets, `execve()`, `ptrace()`, and the vnode pager — the current implementation can be replaced by calls to the `sf_buf` interface.

We describe the implementation of the `sf_buf` interface on the 32-bit *i386* architecture and the 64-bit *amd64* architecture. This implementation reduces the cost of ephemeral mapping management by reusing wherever possible existing virtual-to-physical address mappings. We evaluate the `sf_buf` interface for the pipe, memory disk and networking subsystems. Our results show that these subsystems perform significantly better when using the `sf_buf` interface. On a multiprocessor platform interprocessor interrupts are greatly reduced in number or eliminated altogether.

1 Introduction

Modern operating systems create ephemeral virtual-to-physical mappings for a variety of purposes, ranging from the implementation of interprocess communication to the implementation

of process tracing and debugging. To create an ephemeral mapping two actions are required: the allocation of a temporary kernel virtual address and the modification of the virtual-to-physical address mapping. To date, these actions have been performed through separate interfaces. This paper demonstrates the benefits of combining these actions under a single interface.

This work is motivated by the increasing cost of ephemeral mapping creation, particularly, the increasing cost of modifications to the virtual-to-physical mapping. To see this trend, consider the latency in processor cycles for the `invalpg` instruction across several generations of the IA32 architecture. This instruction invalidates the Translation Look-aside Buffer (TLB) entry for the given virtual address. In general, the operating system must issue this instruction when it changes a virtual-to-physical mapping. When this instruction was introduced in the 486, it took 12 cycles to execute. In the Pentium, its latency increased to 25 cycles. In the Pentium III, its latency increased to ~100 cycles. Finally, in the Pentium 4, its latency has reached ~500 to ~1000 cycles. So, despite a factor of three decrease in the cycle time between a high-end Pentium III and a high-end Pentium 4, the cost of a mapping change measured in wall clock time has actually increased.

Furthermore, on a multiprocessor, the cost of ephemeral mapping creation can be significantly higher if the mapping is shared by two or more processors. Unlike data cache coherence, TLB coherence is generally implemented in software by the operating system [2, 12]: The processor initiating a mapping change issues an interprocessor interrupt (IPI) to each of the processors that share the mapping; the interrupt handler that is executed by each of these processors includes

an instruction, such as `invlpg`, that invalidates that processor's TLB entry for the mapping's virtual address. Consequently, a mapping change is quite costly for all processors involved.

In the past, TLB coherence was only an issue for multiprocessors. Today, however, some implementations of Simultaneous Multi-Threading (SMT), such as the Pentium 4's, require the operating system to implement TLB coherence in a single-processor system.

To reduce the cost and complexity of ephemeral mapping management within an operating system kernel, we introduce the `sf_buf` ephemeral mapping interface. Like Mach's `pmap` interface [11], our objective is to provide a machine-independent interface enabling variant, machine-specific implementations. Unlike `pmap`, our `sf_buf` interface supports allocation of temporary kernel virtual addresses. We describe how various subsystems in the operating system kernel benefit from the `sf_buf` interface.

We present the implementation of the `sf_buf` interface on two representative architectures, *i386*, a 32-bit architecture, and *amd64*, a 64-bit architecture. This implementation is efficient: it performs creation and destruction of ephemeral mappings in $O(1)$ expected time on *i386* and $O(1)$ time on *amd64*. The `sf_buf` interface enables the automatic reuse of ephemeral mappings so that the high cost of mapping changes can be amortized over several uses. In addition, this implementation of the `sf_buf` interface incorporates several techniques for avoiding TLB coherence operations, eliminating the need for costly IPIs.

We have evaluated the performance of the pipe, memory disk and networking subsystems using the `sf_buf` interface. Our results show that these subsystems benefit significantly from its use. For the `bw_pipe` program from the `lmbench` benchmark [10] the `sf_buf` interface improves performance up to 168% on one of our test platforms. In all of our experiments the number of TLB invalidations is greatly reduced or eliminated.

The rest of the paper is organized as follows. The next two sections motivate this work from two different perspectives: First, Section 2 describes the many uses of ephemeral mappings in an operating system kernel; second, Section 3 presents the execution costs for the machine-level operations used to implement ephemeral mappings. We define the `sf_buf` interface and its implementation on two representative architectures in Section 4. Section 5 summarizes the

lines of code reduction in an operating system kernel from using the `sf_buf` interface. Section 6 presents an experimental evaluation of the `sf_buf` interface. We present related work in Section 7 and conclude in Section 8.

2 Ephemeral Mapping Usage

We use FreeBSD 5.3 as an example to demonstrate the use of ephemeral mappings. FreeBSD 5.3 uses ephemeral mappings in a wide variety of places, including the implementation of pipes, memory disks, `sendfile()`, sockets, `execve()`, `ptrace()`, and the vnode pager.

2.1 Pipes

Conventional implementations of Unix pipes perform two copy operations to transfer the data from the writer to the reader. The writer copies the data from the source buffer in its user address space to a buffer in the kernel address space, and the reader later copies this data from the kernel buffer to the destination buffer in its user address space.

In the case of large data transfers that fill the pipe and block the writer, FreeBSD uses ephemeral mappings to eliminate the copy operation by the writer, reducing the number of copy operations from two to one. The writer first determines the set of physical pages underlying the source buffer, then *wires* each of these physical pages disabling their replacement or page-out, and finally passes the set to the receiver through the object implementing the pipe. Later, the reader obtains the set of physical pages from the pipe object. For each physical page, it creates an ephemeral mapping that is private to the current CPU and is not used by other CPUs. Henceforth, we refer to this kind of mapping as a CPU-private ephemeral mapping. The reader then copies the data from the kernel virtual address provided by the ephemeral mapping to the destination buffer in its user address space, destroys the ephemeral mapping, and *unwires* the physical page re-enabling its replacement or page-out.

2.2 Memory Disks

Memory disks have a pool of physical pages. To read from or write to a memory disk a CPU-private ephemeral mapping for the desired pages of the memory disk is created. Then the data is copied between the ephemerally mapped pages

and the read/write buffer provided by the user. After the read or write operation completes, the ephemeral mapping is freed.

2.3 `sendfile(2)` and Sockets

The zero-copy `sendfile(2)` system call and zero-copy socket send use ephemeral mappings in a similar way. For zero-copy send the kernel wires the physical pages corresponding to the user buffer in memory and then creates ephemeral mappings for them. For `sendfile()` it does the same for the pages of the file. The ephemeral mappings persist until the corresponding mbuf chain is freed, e.g., when TCP acknowledgments are received. The kernel then frees the ephemeral mappings and un-wires the corresponding physical pages. These ephemeral mappings are not CPU-private because they need to be shared among all the CPUs — any CPU may use the mappings to retransmit the pages.

Zero-copy socket receive uses ephemeral mappings to implement a form of *page remapping* from the kernel to the user address space [6, 4, 8]. Specifically, the kernel allocates a physical page, creates an ephemeral mapping to it, and injects the physical page and its ephemeral mapping into the network stack at the device driver. After the network interface has stored data into the physical page, the physical page and its mapping are passed upward through the network stack. Ultimately, when an application asks to receive this data, the kernel determines if the application's buffer is appropriately aligned and sized so that the kernel can avoid a copy by replacing the application's current physical page with its own. If so, the application's current physical page is freed, the kernel's physical page replaces it in the application's address space, and the ephemeral mapping is destroyed. Otherwise, the ephemeral mapping is used by the kernel to copy the data from its physical page to the application's.

2.4 `execve(2)`

The `execve(2)` system call transforms the calling process into a new process. The new process is constructed from the given file. This file is either an executable or data for an interpreter, such as a shell. If the file is an executable, FreeBSD's implementation of `execve(2)` uses the ephemeral mapping interface to access the image header describing the executable.

2.5 `ptrace(2)`

The `ptrace(2)` system call enables one process to trace or debug another process. It includes the capability to read or write the memory of the traced process. To read from or write to the traced process's memory, the kernel creates CPU-private ephemeral mappings for the desired physical pages of the traced process. The kernel then copies the data between the ephemerally mapped pages and the buffer provided by the tracing process. The kernel then frees the ephemeral mappings.

2.6 Vnode Pager

The vnode pager creates ephemeral mappings to carry out I/O. These ephemeral mappings are not CPU private. They are used for paging to and from file systems with small block sizes.

3 Cost of Ephemeral Mappings

We focus on the hardware trends that motivate the need for the `sf_buf` interface. In particular, we measure the costs of local and remote TLB invalidations in modern processors. The act of invalidating an entry from a processor's own TLB is called a local TLB invalidation. A remote TLB invalidation, also referred to as TLB shoot-down, is the act of a processor initiating invalidation of an entry from another processor's TLB. When an entry is invalidated from all TLBs in a multiprocessor environment, it is called a global TLB invalidation.

We examine two microbenchmarks: one to measure the cost of a local TLB invalidation and another to measure the cost of a remote TLB invalidation. We modify the kernel to add a custom system call that implements these microbenchmarks. For local invalidation, the system call invalidates a page mapping from the local TLB 100,000 times. For remote invalidation, IPIs are sent to invalidate the TLB entry of the remote CPUs. The remote invalidation is also repeated 100,000 times in the experiment. We perform this experiment on the Pentium Xeon processor and the Opteron processor. The Xeon is an *i386* processor while the Opteron is an *amd64* processor. The Xeon processor implements SMT and has two virtual processors. The Opteron machine has two physical processors. The Xeon operates at 2.4 GHz while the Opteron operates at 1.6 GHz.

For the Xeon the cost of a local TLB invalidation is around 500 CPU cycles when the page table entry (PTE) resides in the data cache, and about 1,000 cycles when it does not. On a Xeon machine with a single physical processor but two virtual processors, the CPU initiating a remote invalidation has to wait for about 4,000 CPU cycles until the remote TLB invalidation completes. On a Xeon machine with two physical processors and four virtual processors, that time increases to about 13,500 CPU cycles.

For the Opteron a local TLB invalidation costs around 95 CPU cycles when the PTE exists in the data cache, and 320 cycles when it does not. Remote TLB invalidations on an Opteron machine with two physical processors cost about 2,030 CPU cycles.

4 Ephemeral Mapping Management

We first present the ephemeral mapping interface. Then, we describe two distinct implementations on representative architectures, *i386* and *amd64*, emphasizing how each implementation is optimized for its underlying architecture. This section concludes with a brief characterization of the implementations on the three other architectures supported by FreeBSD 5.3.

4.1 Interface

The ephemeral mapping management interface consists of four functions that either return an ephemeral mapping object or require one as a parameter. These functions are `sf_buf_alloc()`, `sf_buf_free()`, `sf_buf_kva()`, and `sf_buf_page()`. Table 1 shows the full signature for each of these functions. The ephemeral mapping object is entirely opaque; none of its fields are public. For historical reasons, the ephemeral mapping object is called an `sf_buf`.

`sf_buf_alloc()` returns an `sf_buf` for the given physical page. A physical page is represented by an object called a `vm_page`. An implementation of `sf_buf_alloc()` may, at its discretion, return the same `sf_buf` to multiple callers if they are mapping the same physical page. In general, the advantages of shared `sf_bufs` are (1) that fewer virtual-to-physical mapping changes occur and (2) that less kernel virtual address space is used. The disadvantage is the added complexity of reference counting. The flags argument to `sf_buf_alloc()` is either 0

or one or more of the following values combined with bitwise or:

- “private” denoting that the mapping is for the private use of the calling thread;
- “no wait” denoting that `sf_buf_alloc()` must not sleep if it is unable to allocate an `sf_buf` at the present time; instead, it may return `NULL`; by default, `sf_buf_alloc()` sleeps until an `sf_buf` becomes available for allocation;
- “interruptible” denoting that the sleep by `sf_buf_alloc()` should be interruptible by a signal; if `sf_buf_alloc()`’s sleep is interrupted, it may return `NULL`.

If the “no wait” option is given, then the “interruptible” option has no effect. The “private” option enables some implementations, such as the one for *i386*, to reduce the cost of virtual-to-physical mapping changes. For example, the implementation may avoid remote TLB invalidation. Several uses of this option are described in Section 2 and evaluated in Section 6.

`sf_buf_free()` frees an `sf_buf` when its last reference is released.

`sf_buf_kva()` returns the kernel virtual address of the given `sf_buf`.

`sf_buf_page()` returns the physical page that is mapped by the given `sf_buf`.

4.2 i386 Implementation

Conventionally, the *i386*’s 32-bit virtual address space is split into user and kernel spaces to avoid the overhead of a context switch on entry to and exit from the kernel. Commonly, the split is 3GB for the user space and 1GB for the kernel space. In the past, when physical memories were much smaller than the kernel space, a fraction of the kernel space would be dedicated to a permanent one-to-one, virtual-to-physical mapping for the machine’s entire physical memory. Today, however, *i386* machines frequently have physical memories in excess of their kernel space, making such a *direct* mapping an impossibility.

To accommodate machines with physical memories in excess of their kernel space, the *i386* implementation allocates a configurable amount of the kernel space and uses it to implement a *virtual-to-physical mapping cache* that is indexed by the physical page. In other words, an access to this cache provides a physical page and receives

<code>struct sf_buf *</code>	<code>sf_buf_alloc(struct vm_page *page, int flags)</code>
<code>void</code>	<code>sf_buf_free(struct sf_buf *mapping)</code>
<code>vm_offset_t</code>	<code>sf_buf_kva(struct sf_buf *mapping)</code>
<code>struct vm_page *</code>	<code>sf_buf_page(struct sf_buf *mapping)</code>

Table 1: Ephemeral Mapping Interface

a kernel virtual address for accessing the provided physical page. An access is termed a cache hit if the physical page has an existing virtual-to-physical mapping in the cache. An access is termed a cache miss if the physical page does not have a mapping in the cache and one must be created.

The implementation of the mapping cache consists of two structures containing `sf_buf`s: (1) a hash table of valid `sf_buf`s that is indexed by physical page and (2) an inactive list of unused `sf_buf`s that is maintained in least-recently-used order. An `sf_buf` can appear in both structures simultaneously. In other words, an unused `sf_buf` may still represent a valid mapping.

Figure 1 defines the *i386* implementation of the `sf_buf`. It consists of six fields: an immutable virtual address, a pointer to a physical page, a reference count, a pointer used to implement a hash chain, a pointer used to implement the inactive list, and a CPU mask used for optimizing CPU-private mappings. An `sf_buf` represents a valid mapping if and only if the pointer to a physical page is valid, i.e., it is not `NULL`. An `sf_buf` is on the inactive list if and only if the reference count is zero.

The hash table and inactive list of `sf_buf`s are initialized during kernel initialization. The hash table is initially empty. The inactive list is filled as follows: A range of kernel virtual addresses is allocated by the ephemeral mapping module; for each virtual page in this range, an `sf_buf` is created, its virtual address initialized, and inserted into the inactive list.

The first action by the *i386* implementation of `sf_buf_alloc()` is to search the hash table for an `sf_buf` mapping the given physical page. If one is found, then the next two actions are determined by that `sf_buf`'s `cpumask`. First, if the executing processor does not appear in the `cpumask`, a local TLB invalidation is performed and the executing processor is added to the `cpumask`. Second, if the given flags do not include "private" and the `cpumask` does not include all processors, a remote TLB in-

validation is issued to those processors missing from the `cpumask` and those processors are added to the `cpumask`. The final three actions by `sf_buf_alloc()` are (1) to remove the `sf_buf` from the inactive list if its reference count is zero, (2) to increment its reference count, and (3) to return the `sf_buf`.

If, however, an `sf_buf` mapping the given page is not found in the hash table by `sf_buf_alloc()`, the least recently used `sf_buf` is removed from the inactive list. If the inactive list is empty and the given flags include "no wait", `sf_buf_alloc()` returns `NULL`. If the inactive list is empty and the given flags do not include "no wait", `sf_buf_alloc()` sleeps until an inactive `sf_buf` becomes available. If `sf_buf_alloc()`'s sleep is interrupted because the given flags include "interruptible", `sf_buf_alloc()` returns `NULL`.

Once an inactive `sf_buf` is acquired by `sf_buf_alloc()`, it performs the following five actions. First, if the inactive `sf_buf` represents a valid mapping, specifically, if it has a valid physical page pointer, then it must be removed from the hash table. Second, the `sf_buf`'s physical page pointer is assigned the given physical page, the `sf_buf`'s reference count is set to one, and the `sf_buf` is inserted into the hash table. Third, the page table entry for the `sf_buf`'s virtual address is changed to map the given physical page. Fourth, TLB invalidations are issued and the `cpumask` is set. Both of these operations depend on the state of the old page table entry's accessed bit and the mapping options given. If the old page table entry's accessed bit was clear, then the mapping cannot possibly be cached by any TLB. In this case, no TLB invalidations are issued and the `cpumask` is set to include all processors. If, however, the old page table entry's accessed bit was set, then the mapping options determine the action taken. If the given flags include "private", then a local TLB invalidation is performed and the `cpumask` is set to contain the executing processor. Otherwise, a global TLB invalidation is performed and the `cpumask` is set

to include all processors. Finally, the `sf_buf` is returned.

The implementation of `sf_buf_free()` decrements the `sf_buf`'s reference count, inserting the `sf_buf` into the free list if the reference count becomes zero. When an `sf_buf` is inserted into the free list, a sleeping `sf_buf_alloc()` is awakened.

The implementations of `sf_buf_kva()` and `sf_buf_page()` return the corresponding field from the `sf_buf`.

4.3 *amd64* Implementation

The *amd64* implementation of the ephemeral mapping interface is trivial because of this architecture's 64-bit virtual address space.

During kernel initialization, a permanent, one-to-one, virtual-to-physical mapping is created within the kernel's virtual address space for the machine's entire physical memory using 2MB superpages. Also, by design, the inverse of this mapping is trivially computed, using a single arithmetic operation. This mapping and its inverse are used to implement the ephemeral mapping interface. Because every physical page has a permanent kernel virtual address, there is no recurring virtual address allocation overhead associated with this implementation. Because this mapping is trivially invertible, mapping a physical page back to its kernel virtual address is easy. Because this mapping is permanent there is never a TLB invalidation.

In this implementation, the `sf_buf` is simply an alias for the `vm_page`; in other words, an `sf_buf` pointer references a `vm_page`. Consequently, the implementations of `sf_buf_alloc()` and `sf_buf_page()` are nothing more than cast operations evaluated at compile-time: `sf_buf_alloc()` casts the given `vm_page` pointer to the returned `sf_buf` pointer; conversely, `sf_buf_page()` casts the given `sf_buf` pointer to the returned `vm_page` pointer. Furthermore, none of the mapping options given by the flags passed to `sf_buf_alloc()` requires any action by this implementation: it never performs a remote TLB invalidation so distinct handling for "private" mappings serves no purpose; it never blocks so "interruptible" and "no wait" mappings require no action. The implementation of `sf_buf_free()` is the empty function. The only function to have a non-trivial implementation is `sf_buf_kva()`: It casts an `sf_buf`

pointer to a `vm_page` pointer, dereferences that pointer to obtain the `vm_page`'s physical address, and applies the inverse direct mapping to that physical address to obtain a kernel virtual address.

4.4 Implementations For Other Architectures

The implementations for *alpha* and *ia64* are identical to that of *amd64*. Although the *sparc64* architecture has a 64-bit virtual address space, its virtually-indexed and virtually-tagged cache for instructions and data complicates the implementation. If two or more virtual-to-physical mappings for the same physical page exist, then to maintain cache coherence either the virtual addresses must have the same *color*, meaning they conflict with each other in the cache, or else caching must be disabled for all mappings to the physical page [5]. To make the best of this, the *sparc64* implementation is, roughly speaking, a hybrid of the *i386* and *amd64* implementations: The permanent, one-to-one, virtual-to-physical mapping is used when its color is compatible with the color of the user-level address space mappings for the physical page. Otherwise, the permanent, one-to-one, virtual-to-physical mapping cannot be used, so a virtual address of a compatible color is allocated from a free list and managed through a dictionary as in the *i386* implementation.

5 Using the `sf_buf` Interface

Of the places where the FreeBSD kernel utilizes ephemeral mappings, only three were non-trivially affected by the conversion from the original implementation to the `sf_buf`-based implementation: The conversion of pipes eliminated 42 lines of code; the conversion of zero-copy receive eliminated 306 lines of code; and the conversion of the vnode pager eliminated 18 lines of code. Most of the eliminated code was for the allocation of temporary virtual addresses. For example, to minimize the overhead of allocating temporary virtual addresses, each pipe maintained its own, private cache of virtual addresses that were obtained from the kernel's general-purpose allocator.

```

struct sf_buf {
    LIST_ENTRY(sf_buf) list_entry; /* hash list */
    TAILQ_ENTRY(sf_buf) free_entry; /* inactive list */
    struct          vm_page *m; /* currently mapped page */
    vm_offset_t      kva; /* virtual address of mapping */
    int              ref_count; /* usage of this mapping */
    cpumask_t        cpumask; /* cpus on which mapping is valid */
};

```

Figure 1: The *i386* Ephemeral Mapping Object (*sf_buf*)

6 Performance Evaluation

This section presents the experimental platforms and evaluation of the *sf_buf* interface on the pipe, memory disk and network subsystems.

6.1 Experimental Platforms

The experimental setup consisted of five platforms. The first platform is a Pentium Xeon 2.4 GHz machine, with hyper-threading enabled, having 2 GB of memory. We refer to this platform as Xeon-HTT. Due to hyper-threading the Xeon-HTT has two virtual CPUs on a single physical processor. The next three platforms are identical to Xeon-HTT but have different processor configurations. The second platform runs a uniprocessor kernel resulting in having a single virtual and physical processor. Henceforth, we refer to this platform as Xeon-UP. The third platform has two physical CPUs, each with hyper-threading disabled; we refer to this platform as Xeon-MP. The fourth platform has two physical CPUs with hyper-threading enabled, resulting in having four virtual CPUs. We refer to this platform as Xeon-MP-HTT. Unlike Xeon-UP, multiprocessor kernels run on the other Xeon platforms. The Xeon has an *i386* architecture. Our fifth platform is a dual processor Opteron model 242 (1.6 GHz) with 3 GB of memory. Henceforth, we refer to this platform as Opteron-MP. The Opteron has an *amd64* architecture. All the platforms run FreeBSD 5.3.

6.2 Executive Summary of Results

This section presents an executive summary of the experimental results. For the rest of this paper we refer to the kernel using the *sf_buf* interface as the *sf_buf* kernel and the kernel using the original techniques of ephemeral mapping management as the original kernel. Each experiment is

performed once using the *sf_buf* kernel and once using the original kernel on each of the platforms. For all experiments on all platforms, the *sf_buf* kernel provides noticeable performance improvements.

For the Opteron-MP performance improvement is due to two factors: (1) complete elimination of virtual address allocation cost and (2) complete elimination of local and remote TLB invalidations. Under the original kernel, the machine independent code always allocates a virtual address for creating an ephemeral mapping. The corresponding machine independent code, under the *sf_buf* kernel, does not allocate a virtual address but makes a call to the machine dependent code. The cost of virtual address allocation is avoided in the *amd64* machine dependent implementation of the *sf_buf* interface which returns the permanent one-to-one physical-to-virtual address mappings. Secondly, since the ephemeral mappings returned by the *sf_buf* interface are permanent, all local and remote TLB invalidations for ephemeral mappings are avoided under the *sf_buf* kernel. The above explanation holds true for all experiments on the Opteron-MP and, hence, we do not repeat the explanation for the rest of the paper.

For the various platforms on the Xeon, the performance improvement under the *sf_buf* kernel were due to: (1) reduction of physical-to-virtual address allocation cost and (2) reduction of local and remote TLB invalidations. On the *i386* architecture, the *sf_buf* interface maintains a cache of physical-to-virtual address mappings. While creating an ephemeral mapping under the *sf_buf* kernel, a cache hit results in reuse of a physical-to-virtual mapping. The associated cost is lower than the cost of allocating a new virtual address which is done under the original kernel. Further, a cache hit avoids local and remote TLB invalidations which would have been required under the original kernel. Secondly, if an ephemeral map-

ping is declared CPU-private, it requires no remote TLB invalidations on a cache miss under the `sf_buf` kernel. For each of our experiments in the following sections we articulate the reasons for performance improvement under the `sf_buf` kernel. Unless stated otherwise, the `sf_buf` kernel on a Xeon machine uses a cache of 64K entries of physical-to-virtual address mappings, where each entry corresponds to a single physical page. This cache can map a maximum footprint of 256 MB. For some of the experiments we vary this cache size to study its effects.

The Xeon-UP platform outperforms all other Xeon platforms when the benchmark is single threaded. Only, the web server is multi-threaded, thus only it can exploit symmetric multi-threading (Xeon-HTT), multiple processors (Xeon-MP), or the combination of both (Xeon-MP-HTT). Moreover, Xeon-UP runs a uniprocessor kernel which is not subject to the synchronization overhead incurred by multiprocessor kernels running on the other Xeon platforms.

6.3 Pipes

This experiment used the `lmbench bw_pipe` program [10] under the `sf_buf` kernel and the original kernel. This benchmark creates a Unix pipe between two processes, transfers 50 MB through the pipe in 64 KB chunks and measures the bandwidth obtained. Figure 2 shows the result for this experiment on our test platforms. The `sf_buf` kernel achieved 67%, 129%, 168%, 113% and 22% higher bandwidth than the original kernel for the Xeon-UP, Xeon-HTT, Xeon-MP, Xeon-MP-HTT and the Opteron-MP respectively. For the Opteron-MP, the performance improvement is due to the reasons explained in Section 6.2. For the Xeon platforms, the small set of physical pages used by the benchmark are mapped repeatedly resulting in a near 100% cache-hit rate and complete elimination of local and remote TLB invalidations as shown in Figure 3. For all the experiments in this paper we count the number of remote TLB invalidations issued and not the number of remote TLB invalidations that actually happen on the remote processors.

6.4 Memory Disks

This section presents two experiments — one using Disk Dump (`dd`) and another using the Post-Mark benchmark [9] — to characterize the effect

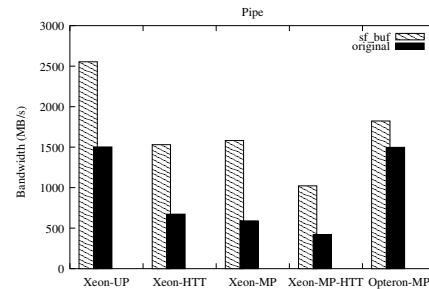


Figure 2: Pipe bandwidth in MB/s

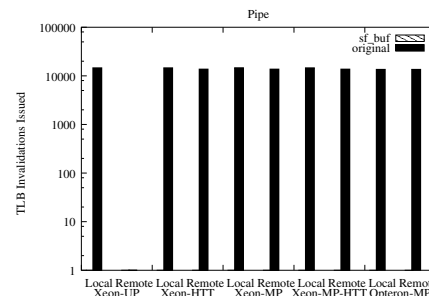


Figure 3: Local and remote TLB invalidations issued for the pipe experiment

of the `sf_buf` interface on memory disks.

6.4.1 Disk Dump (`dd`)

This experiment uses `dd` to transfer a memory disk to the null device using a block size of 64 KB and observes the transfer bandwidth. We perform this experiment for two sizes of memory disks — 128 MB and 512 MB. The size of the `sf_buf` cache on the Xeons is 64K entries, which can map a maximum of 256 MB, larger than the smaller memory disk but smaller than the larger one. Under the `sf_buf` kernel two configurations are used — one using the private mapping option and the other eliminating its use and thus creating default shared mappings.

Figures 4 and 6 show the bandwidth obtained on each of the platforms for the 128 MB disk and 512 MB disk respectively. For the Opteron-MP using the `sf_buf` interface increases the bandwidth by about 37%. On the Xeons, the `sf_buf` interface increases the bandwidth by up to 51%.

Using the private mapping option has no effect on the Opteron-MP because all local and remote TLB invalidations are avoided by the use of permanent, one-to-one physical-to-virtual mappings. Since there is no `sf_buf` cache on the Opteron-MP, similar performance is obtained on both disk sizes.

For the Xeons, the 128 MB disk can be mapped entirely by the `sf_buf` cache causing no local and remote TLB invalidations even when the private mapping option is eliminated. This is shown in Figure 5. Hence, using the private mapping option has negligible effect for the 128 MB disk as shown in Figure 4. However, the 512 MB disk cannot be mapped entirely by the `sf_buf` cache. The sequential disk access of `dd` causes almost a 100% cache-miss under the `sf_buf` kernel. Using the private mapping option reduces the cost of these cache misses by eliminating remote TLB invalidations and thus improves the performance, which is shown in Figure 6. As shown in Figure 7, the use of the private mapping option eliminates all remote TLB invalidations from all Xeon platforms for the 512 MB memory disk.

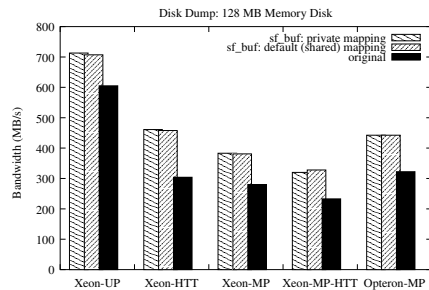


Figure 4: Disk dump bandwidth in MB/s for 128 MB memory disk

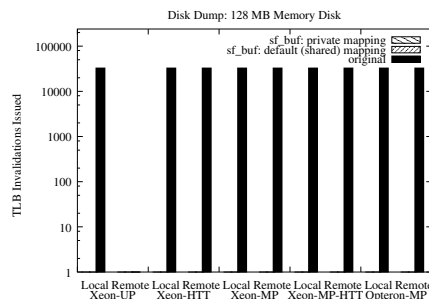


Figure 5: Local and remote TLB invalidations issued for the disk dump experiment on 128 MB memory disk

6.4.2 PostMark

PostMark is a file system benchmark simulating an electronic mail server workload [9]. It creates a pool of continuously changing files and measures the transaction rates where a transaction is creating, deleting, reading from or appending to

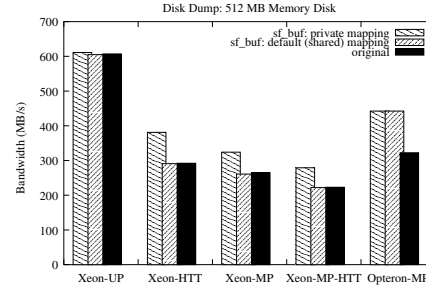


Figure 6: Disk dump bandwidth in MB/s for 512 MB memory disk

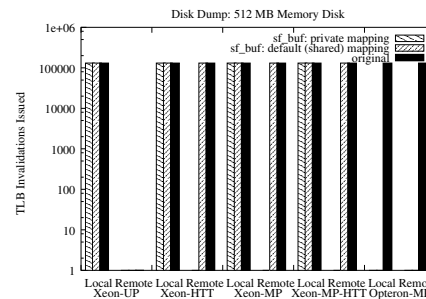


Figure 7: Local and remote TLB invalidations issued for the disk dump experiment on 512 MB memory disk

a file. We used the benchmark's default parameters, i.e., block size of 512 bytes and file sizes ranging from 500 bytes up to 9.77 KB.

We used a 512 MB memory disk for the PostMark benchmark. We used the three prescribed configurations of PostMark. The first configuration has 1,000 initial files and performs 50,000 transactions. The second has 20,000 files and performs 50,000 transactions. The third configuration has 20,000 initial files and performs 100,000 transactions.

PostMark reports the number of transactions performed per second (TPS), and it measures the read and write bandwidth obtained from the system. Figure 8 shows the TPS obtained on each of our platforms for the largest configuration of PostMark. Corresponding results for read and write bandwidths are shown in Figure 9. The results for the two other configurations of PostMark exhibit similar trends and, hence, are not shown in the paper.

For the Opteron-MP, using the `sf_buf` interface increased the TPS by about 11% to about 27%. Read and write bandwidth increased by about 11% to about 17%.

For the Xeon platforms, using the `sf_buf` in-

terface increased the TPS by about 4% to about 13%. Read and write bandwidth went up by about 4% to 15%. The maximum footprint of the PostMark benchmark is about 150 MB under the three configurations used and is completely mapped by the `sf_buf` cache on the Xeons. We did not eliminate the use of the private mapping option on the Xeons for the `sf_buf` kernel as there were no remote TLB invalidations under these workloads. The performance improvement on the Xeons is thus due to the elimination of local and remote TLB invalidations as shown in Figure 10.

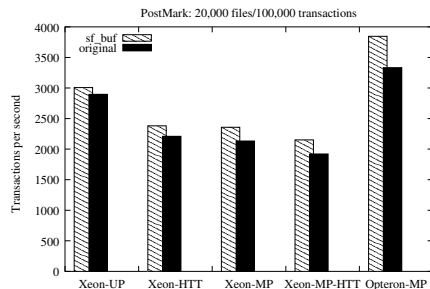


Figure 8: Transactions per second for PostMark with 20,000 files and 100,000 transactions

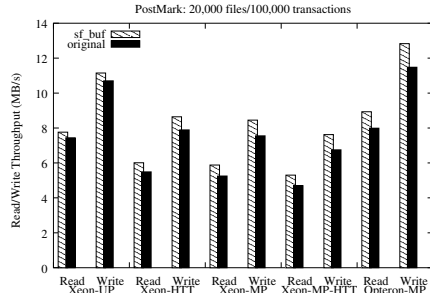


Figure 9: Read/Write Throughput (in MB/s) for PostMark with 20,000 files and 100,000 transactions

6.5 Networking Subsystem

This section uses two sets of experiments — one using `netperf` and another using a web server — to examine the effects of the `sf_buf` interface on the networking subsystem.

6.5.1 Netperf

This experiment examines the throughput achieved between a `netperf` client and server on the same machine. TCP socket send and receive

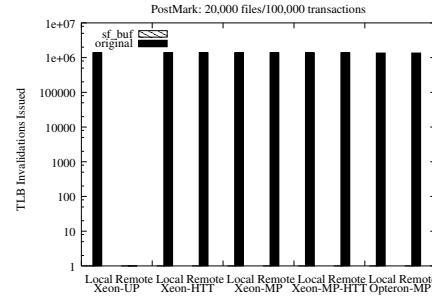


Figure 10: Local and remote TLB invalidations issued for PostMark with 20,000 files and 100,000 transactions

buffer sizes are set to 64 KB for this experiment. Sockets are configured to use zero copy send. We perform two sets of experiments on each platform, one using the default Maximum Transmission Unit (MTU) size of 1500 bytes and another using a large MTU size of 16K bytes.

Figures 11 and 12 show the network throughput obtained under the `sf_buf` kernel and the original kernel on each of our platforms. The larger MTU size yields higher throughput because less CPU time is spent doing TCP segmentation. The throughput improvements from the `sf_buf` interface on all platforms range from about 4% to about 34%. Using the larger MTU size makes the cost of creation of ephemeral mappings a bigger factor in network throughput. Hence, the performance improvement is higher when using the `sf_buf` interface under this scenario.

Reduction in local and remote TLB invalidations explain the above performance improvement as shown in Figures 13 and 14. The `sf_buf` interface greatly reduces TLB invalidations on the Xeons, and completely eliminates them on the Opteron-MP.

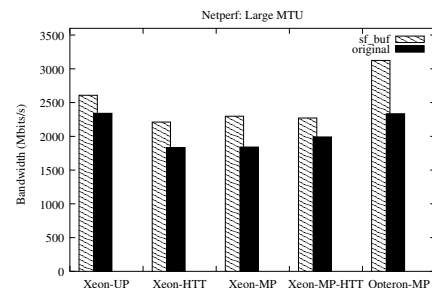


Figure 11: Netperf throughput in Mb/s for large MTU

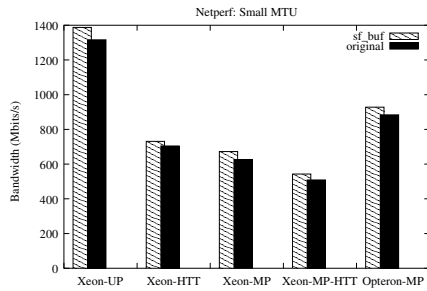


Figure 12: Netperf throughput in Mb/s for small MTU

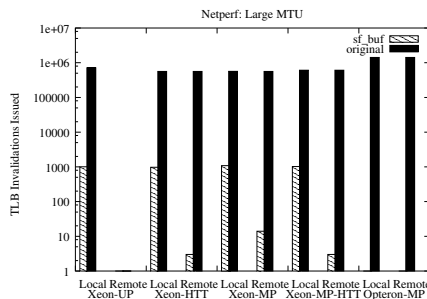


Figure 13: Local and remote TLB invalidations issued for Netperf experiments with large MTU

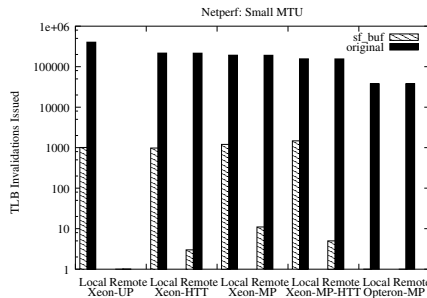


Figure 14: Local and remote TLB invalidations issued for Netperf experiments with small MTU

6.5.2 Web Server

We used apache 2.0.50 as the web server on each of our platforms. We ran an emulation of 30 concurrent clients on a separate machine to generate a workload on the server. The server and client machines were connected via a Gigabit Ethernet link. Apache was configured to use `sendfile(2)`. For this experiment we measure the throughput obtained from the server and count the number of local and remote TLB invalidations on the server machine. The web server was subject to real workloads of web traces from NASA and Rice University's Computer Science Department that have been used in published literature [7, 15]. For the rest of this paper we refer to these workloads as the NASA workload and the Rice workload respectively. These workloads have footprints of 258.7 MB and 1.1 GB respectively.

Figures 15 and 16 show the throughput for all the platforms using both the `sf_buf` kernel and the original kernel for the NASA and the Rice workloads respectively. For the Opteron-MP, the `sf_buf` kernel improves performance by about 6% for the NASA workload and about 14% for the Rice workload. The reasons behind these performance improvements are the same as described earlier in Section 6.2.

For the XeonS, using the `sf_buf` kernel results in performance improvement of up to about 7%. This performance improvement is a result of the reduction in local and remote TLB invalidations as shown in Figures 17 and 18.

For the above experiments the Xeon platforms employed an `sf_buf` cache of 64K entries. To study the effect of the size of this cache on web server throughput we reduced it down to 6K entries. A smaller cache causes more misses, thus increasing the number of TLB invalidations. Implementation of the `sf_buf` interface on the *i386* architecture employs an optimization which avoids TLB invalidations if the page table entry's (PTE) access bit is clear. With TCP checksum offloading enabled, the CPU does not touch the pages to be sent, and as a result the corresponding PTEs have their access bits clear and on a cache miss TLB invalidation is avoided. With TCP checksum offloading disabled, the CPU touches the pages and the corresponding PTEs, causing TLB invalidations on cache misses. So for each cache size we did two experiments, one with TCP checksum offloading enabled and the other by disabling it.

Figure 19 shows the throughput for the NASA workload on the Xeon-MP for the above experiment. For larger cache size slightly higher throughput is obtained because of more reduction in local and remote TLB invalidations as shown in Figure 20. Also, enabling checksum offloading brings local and remote TLB invalidations further down because of the access bit optimization. Reducing the cache size from 64K to 6K entries does not significantly reduce throughput because the hit rate of the ephemeral mapping cache drops from nearly 100% to about 82%. This lower cache hit rate is sufficient to avoid any noticeable performance degradation.

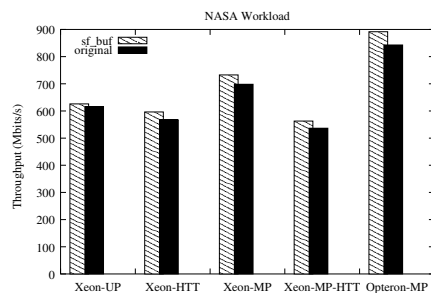


Figure 15: Throughput (in Mbits/s) for the NASA workload

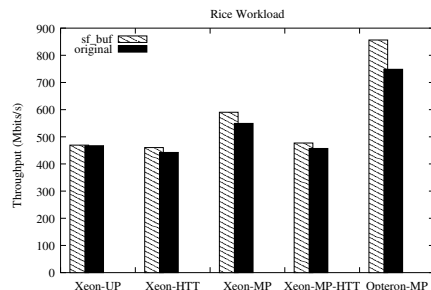


Figure 16: Throughput (in Mbits/s) for the Rice workload

7 Related Work

Chu describes a per process mapping cache for zero-copy TCP in Solaris 2.4 [6]. Since the cache is not shared among all processes in the system its benefits are limited. For example a multi-processed web server using FreeBSD's `sendfile(2)`, like apache 1.3, will not get the maximum benefit from the cache if more than one process transmit the same file. In this case the file pages are the same for all processes so having a common cache would serve best.

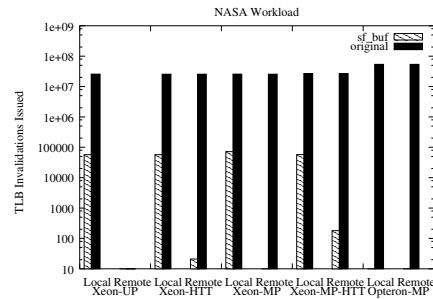


Figure 17: Local and remote TLB invalidations issued for the NASA workload

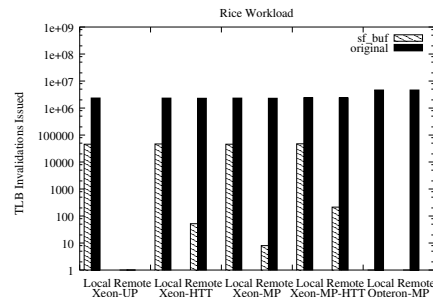


Figure 18: Local and remote TLB invalidations issued for the Rice workload

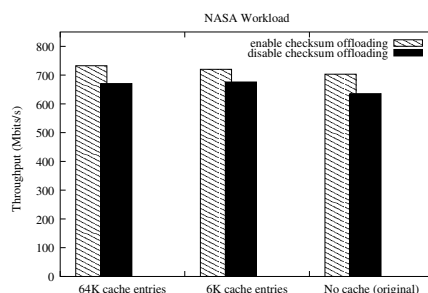


Figure 19: Throughput (in Mbits/s) for the Nasa workload on Xeon-MP with the `sf_buf` cache having 64K or 6K entries and the original kernel and with TCP checksum offloading enabled or disabled

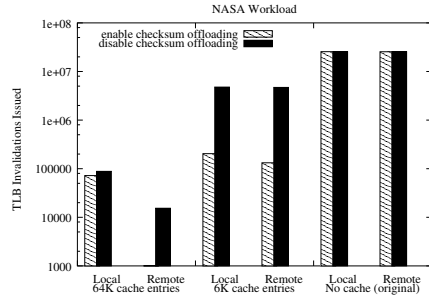


Figure 20: Local and remote TLB invalidations issued for the NASA workload on Xeon-MP with the `sf_buf` cache having 64K or 6K entries and the original kernel and with TCP checksum offloading enabled or disabled

Ruan and Pai extend the mapping cache to be shared among all processes [13]. This cache is `sendfile(2)`-specific. Our work extends the benefits of the shared mapping cache to additional kernel subsystems providing substantial modularity and performance improvements. We provide a uniform API for processor specific implementations. We also study the effect of the cache on multiprocessor systems.

Bonwick and Adams [3] describe Vmem, a generic resource allocator, where kernel virtual addresses are viewed as a type of resource. However, the goals of our work are different from that of Vmem. In the context of kernel virtual address resource, Vmem's goal is to achieve fast allocation and low fragmentation. However, it makes no guarantee that the allocated kernel virtual addresses are "safe", i.e., they require no TLB invalidations. In contrast, the `sf_buf` interface returns kernel virtual addresses that are completely safe in most cases, requiring no TLB invalidations. Additionally, the cost of using the `sf_buf` interface is small.

Bala et al. [1] design a software cache of TLB entries to provide fast access to entries on a TLB miss. This cache mitigates the costs of a TLB miss. The goal of our work is entirely different: to maintain a cache of ephemeral mappings. Because our work re-uses address mappings, it can augment such a cache of TLB entries. This is because the mappings corresponding to the physical pages with entries in the `sf_buf` interface do not need to change in the software TLB cache. In other words, the effectiveness of such a cache (of TLB entries) can be increased with the `sf_buf` interface.

Thekkath and Levy [14] explore techniques for

achieving low-latency communication and implement a low-latency RPC system. They point out re-mapping as one of the sources of the cost of communication. On multiprocessors, this cost is increased due to TLB coherency operations [2]. The `sf_buf` interface obviates the need for re-mapping and hence lowers the cost of communication.

8 Conclusions

Modern operating systems create ephemeral virtual-to-physical mappings for a variety of purposes, ranging from the implementation of inter-process communication to the implementation of process tracing and debugging. The hardware costs of creating these ephemeral mappings are generally increasing with succeeding generations of processors. Moreover, if an ephemeral mapping is to be shared among multiprocessors, those processors must act to maintain the consistency of their TLBs. In this paper we have provided a software solution to alleviate this problem.

In this paper we have devised a new abstraction to be used in the operating system kernel, the ephemeral mapping interface. This interface allocates ephemeral kernel virtual addresses and virtual-to-physical address mappings. The interface is low cost, and greatly reduces the number of costly interprocessor interrupts. We call our ephemeral mapping interface as the `sf_buf` interface. We have described its implementation in the FreeBSD-5.3 kernel on two representative architectures — the *i386* and the *amd64*, and outlined its implementation for the three other architectures supported by FreeBSD. Many kernel subsystems—pipes, memory disks, sockets, `execve()`, `ptrace()`, and the vnode pager—benefit from using the `sf_buf` interface. The `sf_buf` interface also centralizes redundant code from each of these subsystems, reducing their overall size.

We have evaluated the `sf_buf` interface for the pipe, memory disk and networking subsystems. For the `bw_pipe` program of the `lmbench` benchmark [10] the bandwidth improved by up to about 168% on one of our platforms. For memory disks, a disk dump program resulted in about 37% to 51% improvement in bandwidth. For the PostMark benchmark [9] on a memory disk we demonstrate up to 27% increase in transaction throughput. The `sf_buf` interface increases netperf throughput by up to 34%. We also demonstrate tangible benefits for a web server workload

with the `sf_buf` interface. In all these cases, the ephemeral mapping interface greatly reduced or completely eliminated the number of TLB invalidations.

Acknowledgments

We wish to acknowledge Matthew Dillon of the DragonFly BSD Project, Tor Egge of the FreeBSD Project, and David Andersen, our shepherd. Matthew reviewed our work and incorporated it into DragonFly BSD. He also performed extensive benchmarking and developed the implementation for CPU-private mappings that is used on the *i386*. Tor reviewed parts of the *i386* implementation for FreeBSD. Last but not least, David was an enthusiastic and helpful participant in improving the presentation of our work.

References

- [1] K. Bala, M. F. Kaashoek, and W. E. Weihl. Software Prefetching and Caching for Translation Lookaside Buffers. In *First Symposium on Operating Systems Design and Implementation*, pages 243–253, Monterey, California, Nov. 1994.
- [2] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron. Translation lookaside buffer consistency: A software approach. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 113–122, Dec. 1989.
- [3] J. Bonwick and J. Adams. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.
- [4] J. C. Brustoloni and P. Steenkiste. Effects of buffering semantics on i/o performance. In *Operating Systems Design and Implementation*, pages 277–291, 1996.
- [5] R. Cheng. Virtual address cache in Unix. In *Proceedings of the 1987 Summer USENIX Conference*, pages 217–224, 1987.
- [6] H.-K. J. Chu. Zero-copy TCP in Solaris. In *USENIX 1996 Annual Technical Conference*, pages 253–264, Jan. 1996.
- [7] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy asynchronous i/o for event-driven servers. In *USENIX 2004 Annual Technical Conference*, June 2004.
- [8] A. Gallatin, J. Chase, and K. Yocum. Trapeze/IP: TCP/IP at near-gigabit speeds. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, June 1999.
- [9] J. Katcher. Postmark: A new file system benchmark. At http://www.netapp.com/tech_library/3022.html.
- [10] L. McVoy and C. Stalien. Lmbench - tools for performance analysis. At <http://www.bitmover.com/lmbench/>, 1996.
- [11] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.
- [12] B. S. Rosenburg. Low-synchronization translation lookaside buffer consistency in large-scale shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 137–146, Litchfield Park, AZ, Dec. 1989.
- [13] Y. Ruan and V. Pai. Making the Box transparent: System call performance as a first-class result. In *USENIX 2004 Annual Technical Conference*, pages 1–14, June 2004.
- [14] C. A. Thekkath and H. M. Levy. Limits to Low-Latency Communication on High-Speed Networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.
- [15] H. youb Kim, V. S. Pai, and S. Rixner. Increasing Web Server Throughput with Network Interface Data Caching. In *Architectural Support for Programming Languages and Operating Systems*, pages 239–250, San Jose, California, Oct. 2002.

Adaptive Main Memory Compression

Irina Chihaia Tuduce and Thomas Gross

Departement Informatik

ETH Zürich

CH 8092 Zürich, Switzerland

Abstract

Applications that use large data sets frequently exhibit poor performance because the size of their working set exceeds the real memory, causing excess page faults, and ultimately exhibit thrashing behavior.

This paper describes a memory compression solution to this problem that adapts the allocation of real memory between uncompressed and compressed pages and also manages fragmentation without user involvement. The system manages its resources dynamically on the basis of the varying demands of each application and also on the situational requirements that are data dependent. The technique used to localize page fragments in the compressed area allows the system to reclaim or add space easily if it is advisable to shrink or grow the size of the compressed area.

The design is implemented in Linux, runs on both 32-bit and 64-bit architectures, and has been demonstrated to work in practice under complex workload conditions and memory pressure. The benefits from our approach depend on the relationship between the size of the compressed area, the application's compression ratio, and the access pattern of the application. For a range of benchmarks and applications, the system shows an increase in performance by a factor of 1.3 to 55.

1 Introduction

Many applications require more main memory to hold their data than a typical workstation contains. Although the amount of main memory in a workstation has increased with declining prices for semiconductor memories, application developers have even more aggressively increased their demands. Unfortunately, an application must resort to swapping (and eventually, thrashing) when the amount of physical memory is less than what the application (resp. its working set) requires. Substantial disk activity eliminates any benefit that is obtained from increased processor speed. Since the access time of a disk

continues to improve more slowly than the cycle time of processors, techniques to improve the performance of the memory system are of great interest to many applications.

Compression has been used in many settings to increase the effective size of a storage device or to increase the effective bandwidth, and other researchers have proposed to integrate compression into the memory hierarchy. The basic idea of a compressed-memory system is to reserve some memory that would normally be used directly by an application and use this memory region instead to hold pages in compressed form. By compressing some of the data space, the effective memory size available to the applications is made larger and disk accesses are avoided. However, since some of the main memory holds compressed data, the applications have effectively less uncompressed memory than would be available without compression.

The potential benefits of main memory compression depend on the relationship between the size of the compressed area, an application's compression ratio, and an application's access pattern. Because accesses to compressed pages take longer than accesses to uncompressed pages, compressing too much data decreases an application's performance. If an application accesses its data set such that compression does not save enough accesses to disk, or if its pages do not compress well, compression will show no benefit. Therefore, building the core of a system that adaptively finds the size of the compressed area that can improve an application's performance is difficult, and despite its potential to improve the performance of many applications, main memory compression is considered only by few application developers.

This paper presents an adaptive compressed-memory system designed to improve the performance of applications with very large data sets (compression affects only the data area). The adaptive resizing scheme finds the op-

timal size of the compressed area automatically. Because the system must be effective under memory pressure, it uses a simple resizing scheme, which is a function of the number of free blocks in the compressed area (this factor captures an application's access pattern as well). For a set of benchmarks and large applications, measurements show that the compressed area size found by our resizing scheme is among those that improve performance the most.

We allocate and manage the compressed area such that it can be resized easily, without paying a lot to move live fragments around. The key idea is to keep compressed pages in *zones*; the use of zones impose some locality on the blocks of a compressed page, such that the system can easily reclaim or add a zone if it is advisable to shrink or grow the compressed area size.

We examine three simulators that have different access patterns: a model checker, a network simulator, and a car traffic simulator. Depending on their input, the simulators allocate between a few MB and several GB. In this paper, we experiment with inputs that allocate between 164 MB and 2.6 GB. The measurements show that memory compression provides enough memory to these classes of applications to finish their execution on a system with a physical memory smaller than is required to execute without thrashing, and execution proceeds significantly faster than if no compression was used.

Because most of the application developers are mainly interested in a design that works with a stock processor and a commodity PC, we restrict the changes to the software system. The compressed-memory system described here is implemented as a kernel module and patches that hook into the Linux kernel to monitor system activity and control the swap-in and swap-out operations. By restricting the changes to the software system, we can use compression only for those applications that benefit from it. The compressed-memory prototype runs on 32-bit as well as on 64-bit architectures.

Integrating transparent adaptive memory into an operating system raises a number of questions. The design presented here has been demonstrated to work in practice. By choosing a suitable system structure, it is possible to allow the memory system to adapt its size in response to application requirements (an essential property for a transparent system), and by choosing a simple interface to the base operating system, it is possible to limit kernel interaction (essential for acceptance by a user community).

2 Related Work

Several researchers have investigated the use of compression to reduce paging by introducing a new level into the memory hierarchy. The key idea, first sug-

gested by Wilson [15], is to hold evicted pages in compressed form in a compressed area, and intercept page faults to check whether the requested page is available in compressed form before a disk access is initiated. The compressed-memory systems can be classified in software- and hardware-based approaches. Since we want our solution to work with stock hardware, we consider only software-based approaches. For a description of the hardware-based approaches, we refer the interested reader to a study by Alameldeen and Wood [3].

The software-based approaches can be either adaptive or static. The adaptive approaches vary the size of the compressed area dynamically, and are either implementation- or simulation-based investigations. Douglass' early paper [7] adapts the compressed area size based on a global LRU scheme. However, as Kaplan [10] shows later, Douglass' adaptive scheme might have been maladaptive. Douglass implemented his adaptive scheme in Sprite and showed that compression can both improve (up to 62.3%) and decrease (up to 36.4%) an application's performance. Castro et al. [6] adapt the compressed area size depending on whether the page would be uncompressed or on disk if compression was not used. The main drawback of their scheme is that it must analyze every access to the compressed area, and although the approach may work well for small applications, it may not be feasible for large applications with frequent data accesses. The authors implemented their scheme in Linux and report performance improvements of up to 171% for small applications. Wilson and Kaplan [16, 10] resize the compressed area based on recent program behavior. The authors maintain a queue of referenced pages ordered by their recency information. The main drawback of their scheme is that it is based on information (about all pages in the system) that cannot be obtained on current systems; the authors use only simulations to validate their solution. Moreover, as the physical memory size increases, the size of the page queue increases as well, making this approach unsuitable for applications running on systems with large memories.

Static approaches use fixed sizes of the compressed area. Although these studies are useful to assess the benefits of memory compression, they fail to provide a solution that works for different system settings and applications. Cervera et al. [4] present a design implemented in Linux that increases an application's performance by a factor of up to 2 relative to an uncompressed swap system. Nevertheless, on a system with 64 MB physical memory, only 4 MB are allocated to the compressed data, and this small area may not suffice for programs with large working sets. Kjelso et al. [11, 12] use simulations to demonstrate the efficacy of main memory compression. The authors develop a performance model to quantify the performance impact of a software- and hardware-

based compression system for a number of DEC-WRL workloads. Their results show that software-based compression improves system performance by a factor of 2 and hardware-based compression improves performance by up to an order of magnitude.

RAM Doubler is a technology that expands the memory size for Mac OS [2]. It locates small chunks of RAM that applications aren't actively using and makes that memory available to other applications. Moreover, RAM Doubler finds RAM that isn't likely to be accessed again, and compresses it. Finally, if all else fails, the system swaps seldom accessed data to disk. Although RAM Doubler allows the user to open more applications together, the user cannot run applications with memory footprints that exceed the physical memory size. Our work tries to provide enough memory to large applications so that they can run to completion when their memory requirements exceed the physical memory size.

3 Design

A compressed-memory system divides the main memory into two areas: one area holds uncompressed pages and the other area (*compressed area*) holds pages in compressed form. When an application's working set exceeds the uncompressed area size, parts of the data set are compressed and stored in the compressed area. When even the compressed area becomes filled, parts of the compressed data are swapped to disk. On a page fault, the system checks for the faulted page in the compressed area before going to the disk, servicing the page from that area if it is there and saving the cost of a disk access.

The key idea of our compressed-memory design is to organize the compressed area in *zones* of the same size that are linked in a *zone chain*, as shown in Figure 1. As the size of the compressed area grows and shrinks, zones are added and removed from the chain. The system uses a *hash table* for tracking all pages that are stored in the compressed area. If a page is in the compressed area, its entry in the *hash table* points to the zone that stores its compressed data. Moreover, the system uses a global double-linked LRU list for storing the recency information of all compressed pages. *LRU first* and *LRU last* identify the first and last page in the LRU list.

A zone has physical memory to store compressed data and structures to manage the physical memory. A zone's physical data and its structures are allocated/deallocated when a zone is added/deleted. To keep fragmentation as low as possible, a zone's physical memory is divided in blocks of the same size. A compressed page is stored as a list of blocks that are all within the same zone. Each zone uses a *block table* for keeping track of its blocks and their usage information. Furthermore, each zone uses a *comp page table* for mapping compressed pages to their

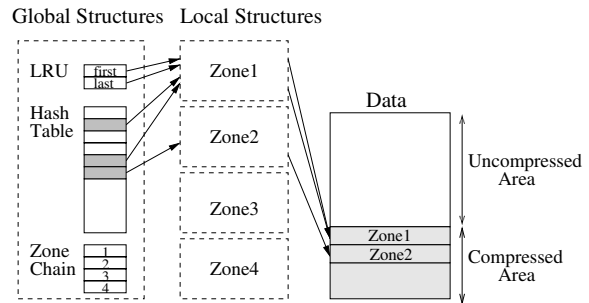


Figure 1: Birdseye view of the compressed-memory system design.

data blocks, as shown in Figure 2. The number of entries in *comp page table* gives the maximum number of compressed pages that can be stored in a zone, and is equal with a *compression factor* multiplied with the number of pages that can be stored in a zone if no compression is used. The following subsections elaborate on how pages are stored and found in the compressed area, as well as how the compressed area is resized.

3.1 Page Insert and Delete

When a page is evicted from the uncompressed area, it is compressed and stored in a compression buffer. The system searches for the first zone that has enough blocks to store the compressed page (the allocation is basically the first-fit algorithm). The system selects a zone from the *zone chain* and uses the *used* field of the *zone structure* to check the number of free blocks in that zone (see Figure 2). If the number of free blocks is insufficient to store the compressed page, another zone is selected and the test is repeated. If the zone has sufficient free blocks, the system uses the *free entry* field of the *zone structure* to select an entry in the *comp page table*. All free entries are linked using the *next* field and the *free entry* field identifies the first element in the list. The selected entry will store information about the new compressed page.

After a zone to store the compressed page is found, the system selects as many blocks as needed to store the compressed data. The system traverses the list of free blocks (whose beginning is identified by the *free block* field) and selects the necessary number of free blocks. All free blocks are linked in a chain by their *next* field in the *block table*. The value of the *free block* field of the *zone structure* is updated to point to the block following the last block selected. The compressed page is now copied into the selected blocks, and the *first* field of the selected entry is set to point to the first block that stores the compressed page. The selected blocks are still linked by their *next* field, and therefore all the blocks that store a compressed page are linked in a chain. The values of the

swap *handle* and the *size* of the compressed page are also set. The *LRU* next and previous fields of the selected entry are now set, and *LRU first* and *LRU last* are updated.

The system computes the index of the new compressed page in the *hash table*. All entries that map to the same index (hash value) are linked in a chain stored in the *next* field of their entries in the *comp page table*. The first element in the chain is identified by the value stored in the *hash table*. The new compressed page is inserted at the beginning of the chain and its *hash table* entry is updated.

On a page fault, the system uses the *hash table* to check whether the faulted page is in the compressed area. If the page is compressed, it is decompressed, its blocks are added to the free list of blocks, its entry in the *comp page table* can be reused, the *zone structure* and the *hash table* entry are updated. If the page is not in the compressed area (does not have an entry in the *hash table*), it is brought from disk into the uncompressed memory.

Because pages are not scattered over multiple zones, when a page is inserted or deleted, the system does not have to keep track of multiple zones that store a page's data. Moreover, when a zone is deleted, the system must not deal with pages that are partially stored in other zones. Therefore, by storing all the blocks of a compressed page within a single zone, we avoid the scatter/gather problem encountered by Douglass [7].

3.2 Interface to the Backing Store

When the compressed area becomes (almost) filled, its *LRU* pages are sent to disk, and the number of free compressed pages is kept above a configurable threshold. Although it is possible to transfer variable-size compressed pages to and from disk, implementing variable-size I/O transfers requires many changes to the OS [7]. To take advantage of the swap mechanism implemented in the OS, we choose to store uncompressed pages on the disk. Moreover, if the page is stored in compressed form, the next time this page is swapped in, it must be first decompressed before it can be used. Therefore, to lower the latency of a future access and to employ the OS swapping services, we decompress a page before sending it to disk.

3.3 Resizing the Compressed Area

The system presented here grows and shrinks the compressed area while applications execute. The resizing decision is based on the amount of data in the compressed area. The system monitors the compressed area utilization. When the amount of memory in the compressed area is above a high threshold, the compressed region is grown by adding a zone. When the amount of memory used is below a low threshold, the compressed area

is shrunk by deleting a zone. As long as the amount of memory used is above the low threshold and below the high threshold, the size of the compressed area remains the same.

All the zones in the system are linked in the *zone chain*, and new zones are added at the end of the chain. When a page is inserted in the compressed area, it is stored in the first zone from the beginning of the *zone chain* that has enough space to store the compressed data. To shrink the compressed area, the system deletes the zone with the smallest number of blocks used (to keep the overhead as low as possible). The compressed pages within the zone to be deleted are relocated within other zones (using again the first-fit algorithm). When the free space within other zones is too small to store the pages to be relocated, some compressed pages are swapped to disk. To grow the compressed area, the system allocates space for a new zone. Because the OS may not have enough free space for the new zone, some uncompressed pages will be compressed and stored in compressed form. At that time, some compressed pages may be swapped to disk to make room for the newly compressed pages. (The *LRU* order is always preserved.)

4 Implementation

In this section, we give an overview of our implementation of the compressed-memory system in Linux. We use Yellow Dog Linux 3.0.1 (YDL) that is built on the 2.6.3 Linux kernel and provides 64-bit support for the Apple G5 machines. The prototype works on both 32-bit and 64-bit architectures, and we installed it on a Pentium 4 PC and on a G5 machine. Although the discussion of our solution is necessarily OS-specific, the issues are general.

Our design is implemented as a loadable module, along with hooks in the operating system to call module functions at specific points. These points are swapping in pages, swapping out pages, and deactivating a swap area. We use a hierarchy of locks and semaphores to protect our code against race conditions. The prototype implements four de/compression algorithms commonly used to de/compress in-memory data: WKdm, WK4x4, LZRW1, and LZ0 [16]. The implementation comprises of about 5,000 lines of C code.

We implemented a performance monitor that collects information about large applications and decides whether to turn on compression. The tool resizes the compressed area dynamically. This tool is implemented in user-space and uses a small library to interact with the kernel module. The implementation of the monitor and library comprises of about 1,200 lines of C code.

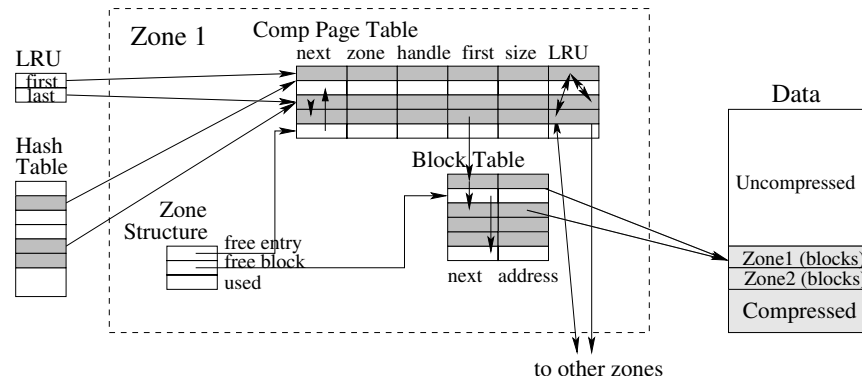


Figure 2: Detailed view of the compressed-memory system design.

4.1 Compressed-Memory Allocation

A kernel module can allocate only kernel memory and is not involved in handling segmentation and paging (since the kernel offers a unified memory management interface to drivers). In Linux, the `kmalloc()` function allocates a memory region that is contiguous in physical memory. Nevertheless, the maximum memory size that can be allocated by `kmalloc()` is 128 KB [13]. Therefore, when dealing with large amounts of memory a module uses the `vmalloc()` function to allocate non-contiguous physical memory in a contiguous virtual address space. Unfortunately, also the memory size that can be allocated by `vmalloc()` is limited, as discussed in the next paragraph.

The Linux kernel splits its address space in two parts: user space and kernel space [8]. On x86 and SPARC architectures, 3 GB are available for processes and the remaining of 1 GB is always mapped by the kernel. (The kernel space limit is 1 GB because the kernel may directly address only memory for which it has set up a page table entry.) From this 1 GB, the first 8 MB are reserved for loading the kernel image to run, as shown in Figure 3. After the kernel image, the `mem_map` array is stored and its size depends on the amount of available memory. In low-memory systems (systems with less than 896 MB), the remaining amount of virtual address space (minus a 2 page gap) is used by the `vmalloc()` function, as shown in Figure 3.a. For illustration, on a Pentium 4 with 512 MB of DRAM, a module can allocate about 400 MB. In high-memory systems, which are systems with more than 896 MB, the `vmalloc` region is followed by the `kmap` region (an area reserved for the mapping of high-memory pages into low memory) and the area for fixed virtual address mappings, as shown in Figure 3.b. On a system with a lot of memory, the size of the `mem_map` array can be significant, and not enough memory is left for the other regions. As the kernel needs these regions, on x86 the `vmalloc` area, the

`kmap` area, and the area for fixed virtual address mapping is defined to be at least 128 MB; this area is denoted by `VMALLOC_RESERVE` at minimum. For illustration, on a Pentium 4 with 1 GB of DRAM, a module can allocate 100 MB. Nevertheless, for applications with large memory footprints, a compressed area of 10% is insufficient. 64-bit architectures aren't as limited in memory usage as 32-bit architectures; a module can allocate 2TB on a 64-bit PowerPC that runs Linux in 64-bit mode.

Because `vmalloc()` is a flexible mechanism to allocate large amounts of data in kernel space, we use `vmalloc()` to allocate memory for the entire compressed area: for the hash table, physical memory, zone structure, comp page table, and block table.

To allow off-line configuration, we have also implemented a compressed-memory system that uses the *bigphysarea* patch [1] to allocate very large amounts of memory to the compressed data. This unofficial patch has been floating around the Net for years; it is so renowned and useful that some distributions apply it to the kernel images they install by default. The patch basically allocates memory at boot time and makes it available to device drivers at runtime. Although boot-time allocation is inelegant and inflexible, it is the only way to bypass the limits imposed by the 32-bit architecture on the size of the `vmalloc` region [13].

4.2 Compressed-Memory Page Daemon

The compressed-memory page daemon (`kcmswapd`) is responsible for swapping out pages, so that we have some free memory in the compressed area. The `kcmswapd` kernel thread is started when memory compression is enabled and is activated on compressed-memory pressure. `kcmswapd` is started after the decision to shrink the compressed area is taken. The daemon swaps out enough compressed pages to make space for the pages stored within the zone to be deleted (these

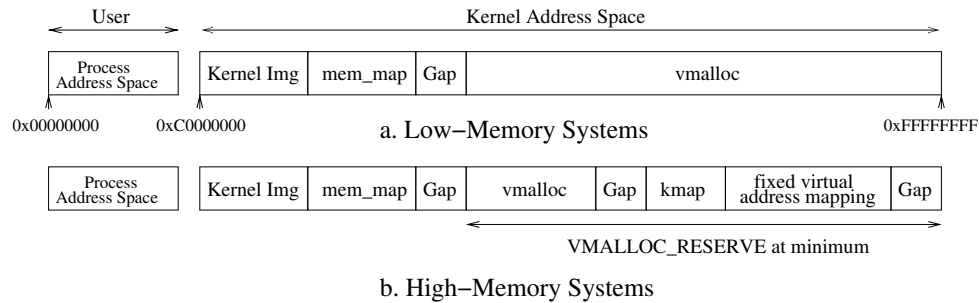


Figure 3: Linux kernel space.

pages have to be relocated within and/or outside the compressed area). Moreover, `kcmswapd` makes space for the uncompressed pages that have to be swapped out to make space for the new zone when the compressed area grows.

5 Evaluation

We select a set of benchmarks and applications that have different memory requirements and access patterns. We conduct a set of experiments to determine how much can a system win from employing memory compression, how much can it lose, and how good the adaptive resizing scheme is.

We use Yellow Dog Linux 3.0.1 (YDL) in 32-bit mode to experiment with benchmarks and applications that run on 32-bit architectures. The system used is a Pentium 4 at 2.6 GHz with a 8 KB L1 data cache, 512 KB L2 cache and 1 GB DRAM; this PC has its swap partition on a IC35L060AVV207-0 ATA disk. Given the memory usage limitations of the 32-bit architectures, to experiment with applications that need compressed areas larger than 100 MB we use an Apple G5 machine that runs YDL in 64-bit mode. The Apple G5 has a dual 64-bit PowerPC 970 microprocessor at 1.8 GHz with a 32 KB L1 data cache, 512 KB L2 cache (per processor) and 1 GB or 1.5 GB DRAM, and has its swap partition on a ST3160023AS ATA disk. For all experiments, we use the WKdm compression algorithm as it shows superior performance over other algorithms [16]. The systems have a block size of 128 bytes, a zone size of 4 MB, and a compression factor of 4. (A compression factor of 4 means that the system can store 4 times more pages within a zone than if no compression was used.)

5.1 Synthetic Benchmarks

The first benchmark shows how much can memory compression degrade system performance. The benchmark, called *thrasher*, pays the cost of compressing pages with-

out gaining any benefit. The benchmark cycles linearly through its working set reading and writing the whole data space. Because Linux uses an LRU algorithm for page replacement, if *thrasher*'s working set doesn't fit in memory, it takes a page fault on each page each time it iterates through the working set. Moreover, each page fault requires a disk read as well as a page write to make room for the faulted page, and we have also the overhead of compressing pages. Because of its access pattern, *thrasher* will always require pages from disk and will never fault on compressed pages. We set the *thrasher*'s working set size to 1.2 GB and we measure its execution time when the size of the physical memory is 1 GB and the compressed area has fixed sizes between 50 MB and 100 MB; *thrasher* has a compression ration of 50% (or 1:2), which is common for many applications. For this set-up the benchmark executes up to 3 times slower than without compression on the Apple G5 and up to 2 times slower on the Pentium 4 PC.

Programs that use dynamic memory allocation access their data through pointers, and hence have irregular access patterns. To investigate the performance of such an application (e.g., written in C++) we use a second benchmark, called *rand*. The advantage of the benchmark over a real application is that its memory footprint and number of data accesses can be changed easily. The benchmark reads and writes its data set randomly and has a compression ration of 50%. We consider three variants that allocate 1.2 GB, 1.4 GB, and 1.8 GB and access their data sets 200,000, 1,200,000, and 6,000,000 times; we execute the benchmark on an Apple G5 with 1 GB physical memory. The three variants finish execution in 538.62 sec, 5,484.75 sec, and 47,617.38 sec. When we apply our adaptive compression technique to these variants their performance improves by a factor of 3.66, 11.88, and 18.96; the compressed area size found by the resizing scheme is 64 MB, 96 MB, and 140 MB.

Memory available	W/o compr.		W/ compr.	
	sec	slowdown	sec	speedup
100%	6	-	6	-
97%	10	1.66	16	0.62
92%	16	2.66	65	0.24
87%	403	67.16	391	1.03
85%	1,307	217.83	450	3.22
82%	2,431	405.16	791	3.07
80%	3,601	600.16	1,175	3.06
78%	4,645	774.16	1,433	3.24
75%	5,649	941.50	1,609	3.51
73%	8,789	1,464.83	2,177	4.03

Table 1: Execution time of *nodes_2_4_3* model on a Pentium 4 PC at 2.6 GHz.

Memory available	W/o compr.		W/ compr.	
	sec	slowdown	sec	speedup
100%	10	-	10	-
97%	37	3.7	32	1.15
92%	47	4.7	71	0.66
87%	290	29	1,045	0.27
85%	1,212	121	1,270	0.95
82%	2,165	216	1,382	1.56
80%	3,290	329	1,508	2.18
78%	4,900	490	1,670	2.93
75%	5,650	565	1,931	2.92
73%	6,780	678	2,233	3.03

Table 2: Execution time of *nodes_2_4_3* model on an Apple G5 at 1.8 GHz.

5.2 Applications

5.2.1 Symbolic Model Verifier

SMV is a method based on Binary Decision Diagrams (BDDs) used for formal verification of finite state systems. We use Yang’s SMV implementation since it demonstrated superior performance over other implementations [17]. We choose different SMV inputs that model the FireWire protocol [14]. SMV’s working set is equal to its memory footprint (SMV uses all the memory it allocates during its execution rather than a small subset) and has a compression ratio of 52% on average.

We consider an SMV model, *nodes_2_4_3*, with a small memory footprint of 164 MB, to explore the limitations of compressed memory. An application with such a small footprint is unlikely to require compression but allows us to perform many experiments. We conduct the first set of experiments on the Pentium 4 PC at 2.6 GHz. We configure the system such that the amount of memory available is 97% to 73% of memory allocated. The measurements are summarized in Table 1, column “W/o compr.” and show that when physical memory is smaller than SMV’s working set, SMV’s performance is degraded substantially. In the next set of experiments, SMV executes on the adaptive compressed-memory system. The measurements are summarized in Table 1, column “W/ compr.”, and indicate that when the amount of memory available is 87% to 73% of memory allocated, our adaptive compression technique increases performance by a factor of up to 4. The measurements also show that for this small application when the memory shortage is not big enough (memory available is 97% to 92% of memory allocated), taking away space from the SMV model for the compressed area will slowdown the application.

We repeat the experiments on the Apple G5 that has a different architecture and a 1.8 GHz processor, and

we summarize the results in Table 2. (Different DRAM chips we use have a negligible influence of 0.02% on an application’s performance.) The results for the adaptive set-up, summarized in column “W/ compr.”, indicate that when SMV executes on the G5 machine with the compressed-memory system described here, SMV’s performance improves by a factor of up to 3. Overall, the results indicate that on a slow machine (Apple G5), compression improves performance for a smaller range of configurations than on a fast machine (Pentium 4 at 2.6 GHz). The measurements confirm other researchers’ results: on older machines memory compression can increase system performance by a factor of up to 2 relative to an uncompressed swap system [4, 12, 6]. Moreover, our measurements show that memory compression becomes more attractive as the processor speed increases.

5.2.2 NS2 Network Simulator

NS2 is a network simulator used to simulate different protocols over wired and wireless networks. We choose different inputs that simulate the AODV protocol over a wireless network. NS2’s working set is smaller than its memory footprint (it uses only a small subset of its data at any one time) and has a compression ratio of 20% (or 1:5) on average. The amount of memory allocated by a NS2 simulation is determined by the number of nodes simulated, and the size of the memory used is given by the number of traffic connections that are simulated.

We consider two simulations that allocate 880 MB and 1.5 GB. We configure the system such that the amount of memory provided is less than memory allocated, and we measure the simulations’ execution time and compute their slowdown. The results are summarized in Table 3, column “W/o compr.”, and show that when memory available is 68% to 43% of memory allocated, NS2 executes slightly slower than normal. When we apply

Working set size	Memory available	W/o compr. sec	W/ compr. sec	speedup
Pentium 4 at 2.6 GHz				
730 MB	70%	145	128	1.13
880 MB	58%	205	168	1.22
990 MB	51%	275	197	1.39
G5 at 1.8 GHz				
730 MB	70%	243	226	1.07
880 MB	58%	345	252	1.36
990 MB	51%	398	319	1.23

Table 4: NS2 execution time on an adaptive compressed-memory system.

our compression technique to NS2 executing with the same reduced memory allocation, its performance improves by a factor of up to 1.4. The measurements for the adaptive set-up are summarized in Table 3, column “W/ compr.”. The results show that because NS2 allocates a large amount of data but uses only a small subset of its data at any one time, compression does not improve performance much, but fortunately, compression does not hurt either.

The second set of experiments uses inputs that allocate 730 MB, 880 MB, and 990 MB. We execute the selected simulations on a system with and without compression when memory available is 70%, 68%, and 51% of memory allocated, and we summarize the results in Table 4. The data in column “W/ compr.” show that because NS2’s working set is small (smaller than memory allocated) and fits into small memories, compression does not improve NS2’s performance much. Overall, the measurements show that on the faster Pentium 4 PC compression improvements are slightly bigger than on the slower G5 machine.

5.2.3 *qsim* Traffic Simulator

qsim [5] is a motor vehicle traffic simulator that employs a queue to model the behavior of varying traffic conditions. Although a simulation can be distributed on many computers (e.g., a cluster), the simulation requires hosts with memory sizes bigger than 1 GB. For a geographic region, the number of travelers (agents) simulated determine the amount of memory allocated to the simulation and the number of (real) traffic hours being simulated gives the execution time of the simulation.

We consider simulations that allocate 1.3 GB, 1.7 GB, 1.9 GB, and 2.6 GB and simulate the traffic on the road network of Switzerland. We measure the execution time of these simulations on the G5 machine without compression and with adaptive compression, and we summarize the results in Table 5. The system has a block size

of 128 bytes, a zone size of 4 MB, and a compression factor of 9. The results in column “W/ compr.” show that when *qsim* executes on our compressed-memory system, its performance improves by a factor of 20 to 55. *qsim*’s working set is equal to its memory footprint (during its execution, *qsim* uses all the memory it allocates), and has a compression ratio of 10% (or 1:10) on average. Because *qsim* compresses so well, even when the amount of memory provided is much smaller than memory allocated, the simulation fits into the uncompressed and compressed memory and finishes its execution in a reasonable time. For instance, although the last simulation listed in Table 5 allocates 2.6 GB, it succeeds to finish its execution on a system with only 1 GB physical memory, and this would not be possible without compression.

We repeat the simulation that allocates 1.9 GB on the Pentium 4 PC with 1 GB physical memory. On an Apple G5 with 1 GB physical memory the system allocates 140 MB to the compressed data, but on the Pentium 4 PC the compressed area can be 100 MB at most. The measurements show that because the Pentium 4 PC fails to allocate enough memory to the compressed data, the simulation executes 8.5 times slower than on the Apple G5 (although the Pentium 4 processor is faster than the PowerPC processor). This experiment shows the importance of a flexible OS support: if the amount of memory that can be allocated in kernel mode was not limited, main memory compression would improve the performance of this large application considerably.

5.3 Discussion

Our analysis examines the performance of three applications and shows that compression improves the performance for all these applications, but varies according to the memory access behavior and also to the compression ratio employed.

SMV and *qsim* use their entire working set during the execution. When the amount of memory provided is less than memory allocated by 10% or more, SMV executes approximately 600 times slower than without swapping. The measurements show that when the amount of memory available is 15% smaller than SMV’s working set, our compression technique provides an increase in performance by a factor of 3 to 4 depending on the processor used (a factor 3 for a G5 and 4 for a Pentium 4). When we apply our compression techniques to *qsim* its execution is improved by a factor of 20 to 55.

The NS2 simulator allocates a large amount of data but uses only a small subset of its data at any one time, and thus provides an example that is much different from SMV. Under normal execution (without the aid of our compression techniques) when physical memory is 40% smaller than memory allocated, NS2’ execution is slow-

Working set size	Memory available	W/o compr.		W/ compr.	
		sec	slowdown	sec	speedup
880 MB	58%	345	1.36	252	1.36
	50%	426	1.69	313	1.36
	43%	586	2.32	425	1.37
1.5 GB	68%	1,335	1.11	1,275	1.04
	62%	1,351	1.12	1,215	1.11

Table 3: NS2 execution time on an Apple G5 at 1.8 GHz.

Working set size	Physical memory	W/o compr. sec	W/ compr.	
			sec	speedup
1.3 GB	1 GB	3,993	135.45	29.47
1.7 GB	1 GB	24,580	513.66	47.85
	1.5 GB	2,900	141.53	20.49
1.9 GB	1 GB	46,049	825.72	55.76
	1.5 GB	11,456	277.91	41.22
2.6 GB	1 GB	51,569	988.01	52.19
	1.5 GB	13,319	332.50	40.05

Table 5: *qsim* execution time on an Apple G5 at 1.8 GHz.

down by a factor of up to 2. When we apply our compression techniques to NS2 executing with the same reduced memory allocation its performance improves by a factor of up to 1.4.

5.4 Adaptivity

Previous work determines the amount of data to be compressed by monitoring every access to the compressed data [10, 16, 6]. The system keeps track of the pages that would be anyway in memory (with and without compression) and pages that are in (compressed) memory only because compression is turned on. The decision to shrink or grow the compressed area is based on the number of accesses to these two types of (compressed) pages. This approach succeeds to detect when the size of the compressed area should be zero, which is not the case with our resizing scheme (see Table 1, column “W/ compr.” when memory available is 97% and 92% of memory allocated and Table 2, column “W/ compr.” when memory available is 92%, 87%, and 85% of memory allocated). Nevertheless, for each access to the compressed data, the system has to check whether the page would be in memory if compression was turned off. To check this, the system has to find the position of the page in the (LRU) list of all compressed pages. Because to search a list of n pages takes $O(n)$ time in the worse case, this approach is not feasible for applications with large data sets. We experimented with schemes that monitor each access to the compressed data, and we found that the check operations

decrease system’s performance by a factor of 20 to 30. To summarize, although previous resizing schemes succeed to detect when compression should be turned off, they cannot be used for applications with large data sets.

We take a different approach and adapt the compressed area size such that the uncompressed and compressed memory contain most of an application’s working set. (For this scenario most of the application’s disk accesses are avoided.) Our approach is based on the observation that when the compressed area is larger than an application’s memory footprint, some space within the compressed area is unused. By default, compression is turned off and the system checks the size of memory available periodically. If an application’s memory needs exceed a certain threshold, compression is turned on for that application and a zone is added to the compressed area. From now on, the system checks the amount of compressed data periodically and decides whether to change the size of the compressed area.

If the amount of free memory in the compressed area is bigger than the size of four zones, the compressed area is shrunk by deleting a zone. If not, the system checks whether the size of the free memory is smaller than the size of a zone; if so, a new zone is added to the compressed area. As long as the size of free compressed memory is between the size of a zone and four zones the compressed area size remains the same; using this strategy we avoid resizing the compressed area too often. The values of the shrink and grow threshold are sensitive to the size of an application’s working set: small

applications that execute on systems with small memories require small threshold values (the compressed areas they require are small). For the (large) applications we selected, we experimented with values of the shrink threshold of three and four zones, and we found that the performance improvements are the same. Furthermore, when the value of the shrink threshold is bigger than the size of four zones, the degree at which performance is improved decreases. The decrease in performance is because the size of the compressed area found by the adaptive scheme grows due to the increase in free compressed memory.

To assess the accuracy of our adaptation scheme, we examine the performance of the *qsim* simulator and *rand* benchmark on a system with fixed sizes of the compressed area and on an adaptive compressed-memory system. We choose these two applications because they have different memory access behavior, different compression ratio, require large sizes of the compressed area, and finish execution in a reasonable time. We run the experiments on the G5 machine that has a block size of 128 bytes and a zone size of 4 MB; the value of the compression factor is 9 for *qsim* and 14 for *rand*. The measurements for the *qsim* simulations and for the *rand* benchmarks are summarized in Figure 4 and Figure 5, and show that the size found by our resizing scheme is among those that improve performance the most.

To sum up, our design and adaptation scheme minimize the number of resizing operations: the memory system usage is checked periodically (and not at every access to the compressed data), the compressed area is not resized every time the system usage is checked, and the compressed area is grown and shrunk by adding and removing zones (and not single pages).

6 Design Tradeoffs

System performance often depends upon more than one factor. In this section we isolate the performance effects of each factor that influence the compressed-memory overhead. We use the 2^k design to determine the effect of k factors, each of which has two design alternatives. We use the 2^k design because it is easy to analyze and helps sorting out factors in the order of impact [9].

As previously described, the compressed area is based on zones that are self contained consisting of all necessary overhead data structures required to manage the compressed memory within a zone. Because a zone uses the *block table* and *comp page table* to manage its compressed data (see Figure 2), the compressed-memory overhead is the sum of the sizes of these two data structures: $overhead = sizeof(BlockTable) + sizeof(CompPageTable)$. (Because all zones are equal in size, all *block tables* and *comp page tables* have the

Factor	Level -1	Level 1
Compr factor	4	14
Block size	64 B	1024 B
Zone size	2 MB	8 MB

Table 6: Factors and levels.

same size.) Formally, the memory overhead is given by Eq. 1. (The number of entries in the *comp page table* gives the maximum number of compressed pages that can be stored within a zone, and can be changed by changing the *compression factor* parameter.)

Eq. 1 shows that the three factors that affect the compressed-memory overhead and need to be studied are the compression factor (*ComprFactor*), block size (*BlockSize*), and zone size (*ZoneSize*); the page size factor (*PageSize*) is fixed. We use the 2^k factorial design to determine the effect of the three factors ($k=3$) on an application's execution time [9]. The factors and their level assignments for the *qsim* simulations are shown in Table 6. The 2^k design and the measured performance in *sec* is shown in Table 7. We use the sign table method to compute the portion of variation explained by the three factors and their interaction, and we summarize the computations in Table 8, column "*qsim* simulations". The results show that most of the variation in the performance of the *qsim* application is explained by the compression factor (column "ComprFactor") and the interaction between the compression factor and block size (column "ComprFactor+BlockSize"). Moreover, for the large simulations, the measurements indicate that a compressed-memory system with a small zone size decreases the performance considerably (the zone size explains more than 30% of the variation). We use the same 2^k design to determine the effect of the three factors on *rand* benchmark performance; the only difference is that the two levels of the compression factor are 4 and 20. The measurements summarized in Table 8, column "*rand* benchmark", show again that the most important factors are the compression factor (column "ComprFactor") and the interaction between the compression factor and block size (column "ComprFactor+BlockSize"). The results also show that large applications require large zone sizes.

Let us consider an application with a high compression ratio that executes on a system with a small compression factor. Because the number of entries in the *comp page table* is smaller than the number of compressed pages that can be stored in a zone, some memory remains unused. On the other hand, a high value of the compression factor increases the size of the *comp page table* unnecessarily. The measurements summarized in Figure 6(a) show that a compressed-memory system improves an application's performance when its compression factor is

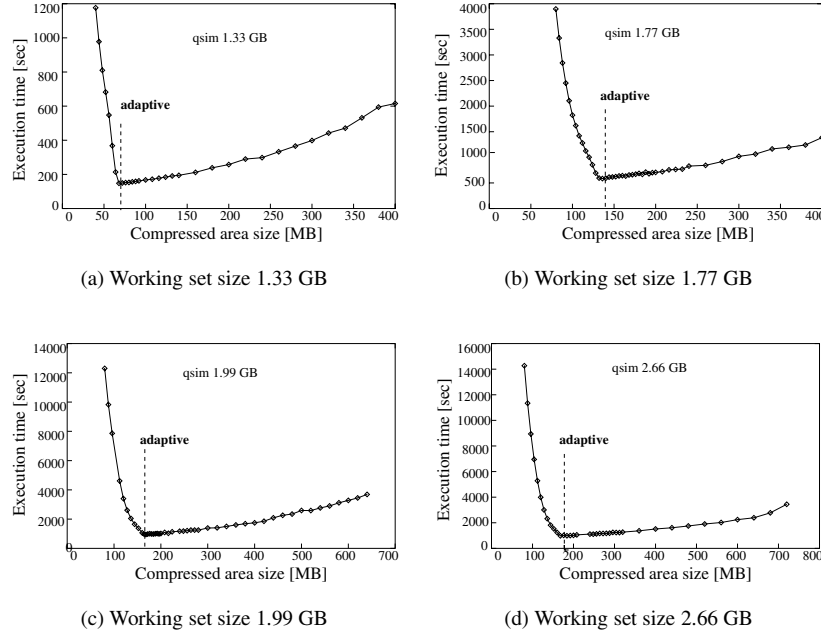


Figure 4: *qsim* execution time on an Apple G5 with 1 GB physical memory for fixed sizes of the compressed area.

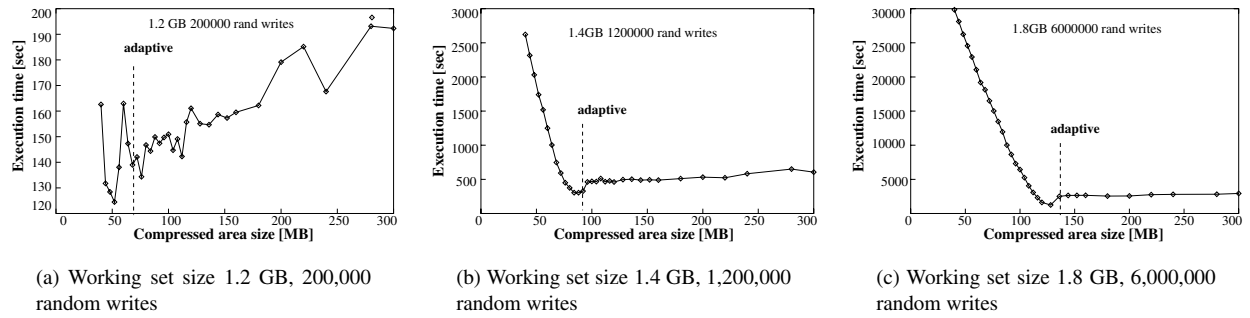


Figure 5: *rand* execution time on an Apple G5 with 1 GB physical memory for fixed sizes of the compressed area.

$$overhead = 2 \cdot \text{sizeof}(\text{int}) \cdot \frac{\text{ZoneSize}}{\text{BlockSize}} + \text{sizeof}(\text{comp_page_entry}) \cdot \text{ComprFactor} \cdot \frac{\text{ZoneSize}}{\text{PageSize}} \quad (1)$$

Test	Compr factor	2 MB		8 MB	
		64 B	1024 B	64 B	1024 B
1.33 GB	4	245.30	258.30	688.96	165.64
	14	147.38	253.49	144.51	152.63
1.77 GB	4	2,229.04	1,796.98	2,803.01	552.47
	14	595.96	1,980.51	551.54	660.04
1.99 GB	4	7,351.71	5,479.42	4,395.01	888.60
	14	954.99	6,354.63	872.34	973.74
2.66 GB	4	7,721.11	6,340.01	3,688.07	1,055.76
	14	1,116.68	7,380.94	981.45	1,092.29

Table 7: Results of the 2^k experiment. The performance of different *qsim* simulations is measured in *sec* on an Apple G5 with 1 GB physical memory.

	<i>qsim</i> simulations				<i>rand</i> benchmark		
	1.33 GB	1.77 GB	1.99 GB	2.66 GB	1.2 GB	1.4 GB	1.8 GB
ComprFactor	23.61%	27.91%	18.50%	13.13%	33.34%	17.58%	6.01%
BlockSize	8.51%	3.06%	0.00%	1.08%	24.75%	7.46%	5.84%
ZoneSize	3.31%	8.96%	39.03%	47.99%	2.68%	22.93%	58.38%
ComprFactor+BlockSize	21.14%	37.69%	27.29%	20.90%	28.66%	36.56%	10.42%
ComprFactor+ZoneSize	11.2%	1.04%	1.00%	1.62%	1.20%	3.21%	5.25%
BlockSize+ZoneSize	21.81%	20.70%	11.08%	10.62%	1.78%	8.46%	6.00%
ComprFactor+ +Block+ZoneSize	10.42%	0.64%	3.10%	4.65%	7.58%	3.81%	8.10%

Table 8: The portion of variation explained by the three factors and their interaction.

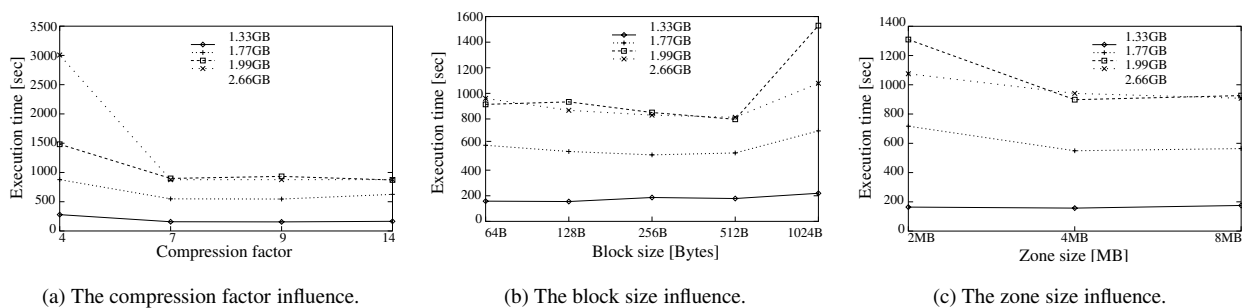


Figure 6: The influence of the three factors on the *qsim* performance. The simulations run on an Apple G5 with 1 GB physical memory.

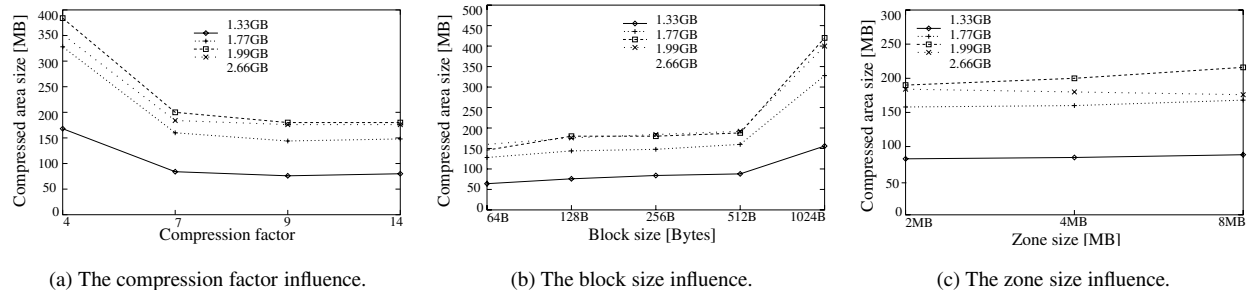


Figure 7: The influence of the three factors on the size of the compressed area. The simulations run on an Apple G5 with 1 GB physical memory.

equal to or bigger than an application's compression ratio. Furthermore, the data in Figure 7(a) indicates that when the compression factor is smaller than an application's compression ratio, also the size of the compressed area is bigger than that which would suffice if enough entries to address a zone's memory were available.

The internal fragmentation of the compressed-memory system is the sum of the unused space in the last block of each compressed page. Because the percentage of unused memory in the last block increases when the block size increases, the internal fragmentation increases as well. The measurements summarized in Figure 6(b) show that block sizes smaller than 512 bytes yield good performance improvements, and a block size of 1024 bytes decreases the *qsim* performance for all input sizes. The data in Figure 7(b) indicate that also the size of the compressed area is influenced by the degree of the internal fragmentation.

The results in Figure 6(c) indicate that a zone size of 4 MB improves *qsim* performance for the simulations that allocate 1.33 GB and 1.77 GB, but zones larger than 4 MB are required for the simulations with large data footprints (those that allocate 1.99 GB and 2.66 GB). Moreover, the data in Figure 7 show that when the compressed area is allocated in zones of big sizes, the amount of compressed area grows slightly because of the zone granularity.

To summarize, our analysis shows that a compressed-memory system that has a high value of the compression factor will improve performance for a wide range of applications (with different compression ratio). Measurements indicate that block sizes smaller than 512 bytes work well for the selected applications. Furthermore, as the size of an application's working set increases, also the zone size should increase for compression to show maximum performance improvements.

7 Concluding Remarks

This paper describes a transparent and effective solution to the problem of executing applications with large data sets when the size of the physical memory is less than what is required to run the application without thrashing. Without a compressed-memory level in the memory hierarchy, such applications experience memory starvation. We describe a practical design for an adaptive compressed-memory system and demonstrate that it can be integrated into an existing general-purpose operating system. The key idea is to keep compressed pages in zones; zones impose some locality on the blocks of a compressed page so that at a later time, the operating system is able to reclaim a zone if it is advisable to shrink the size of the compressed data.

We evaluated the effectiveness of our system on a range of benchmarks and applications. For synthetic benchmarks and small applications we observe a slowdown up to a factor of 3; further tuning may further reduce this penalty. For realistic applications, we observe an increase in performance by a factor of 1.3 to 55. The dramatic improvements in performance are directly correlated to the memory access patterns of each program. If the working set and memory footprint are strongly correlated, our compression technique is more effective because the effects of memory starvation are more critical to the program's overall performance. If the working set is a small subset of the memory footprint, memory compression improves performance but since memory starvation imposes a smaller impact on program execution, its benefit is seen only during those periods of memory starvation. The main memory compression benefits are sustained under complex workload conditions and memory pressure, and the overheads are small.

Although the amount of main memory in a workstation has increased with declining prices for semiconductor memories, application developers have even more

aggressively increased their demands. A compressed-memory level is a beneficial addition to the classical memory hierarchy of a modern operating system, and this addition can be provided without significant effort. The compressed-memory level exploits the tremendous advances in processor speed that have not been matched by corresponding increases in disk performance. Therefore, if access times to memory and disk continue to improve over the next decade at the same rate as they did during the last decade (the likely scenario), software-only compressed-memory systems are an attractive approach to improve total system performance.

Acknowledgements

We thank Kai Nagel for an version of the *qsim* traffic simulator. We appreciate feedback and comments by the reviewers.

This work was funded, in part, by the NCCR “Mobile Information and Communication Systems”, a research program of the Swiss National Science Foundation, and by a gift from Intel’s Microprocessor Research Laboratory.

References

- [1] Bigphysarea. <http://www.polyware.nl/middelink/En/hob-v4l.html#bigphysarea>.
- [2] RAM Doubler 8. <http://www.powerbookcentral.com/features/ramdoubler.shtml>.
- [3] A. Alameldeen and D. Wood. Adaptive Cache Compression for High-Performance Processors. In *Proc. ISCA*, pages 212–223, Munich, Germany, June 2004. IEEE.
- [4] R. Cervera, T. Cortes, and Y. Becerra. Improving Application Performance through Swap Compression. In *Proc. 1999 USENIX Tech. Conf.: FREENIX Track*, pages 207–218, Monterey, CA, June 1999.
- [5] N. Cetin, A. Burri, and K. Nagel. A Large-Scale Multi-Agent Traffic Microsimulation based on Queue Model. In *STRC*, Monte Verita, Switzerland, March 2003.
- [6] R. de Castro, A do Lago, and D. Da Silva. Adaptive Compressed Caching: Design and Implementation. In *Proc. SBAC-PAD*, pages 10–18, Sao Paulo, Brazil, Nov. 2003. IEEE.
- [7] F. Dougliis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *Proc. Winter USENIX Conference*, pages 519–529, San Diego, CA, Jan. 1993.
- [8] M. Gorman. Understanding The Linux Virtual Memory Manager. Master’s thesis, University of Limerick, July 2003. <http://www.skynet.ie/mel/projects/vm/>.
- [9] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, New York, April 1991.
- [10] S. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation*. PhD thesis, The University of Texas at Austin, Dec. 1999.
- [11] M. Kjelso, M. Gooch, and S. Jones. Design and Performance of a Main Memory Hardware Compressor. In *Proc. 22nd Euromicro Conf.*, pages 423–430. IEEE Computer Society Press, Sept. 1996.
- [12] M. Kjelso, M. Gooch, and S. Jones. Performance Evaluation of Computer Architectures with Main Memory Data Compression. *Journal of Systems Architecture*, 45:571–590, 1999.
- [13] A. Rubini and J. Corbet. *Linux Device Drivers*. O’Reilly, 2nd edition, June 2001.
- [14] V. Schuppan and A. Biere. A Simple Verification of the Tree Identify Protocol with SMV. In *Proc. IEEE 1394 (FireWire) Workshop*, pages 31–34, Berlin, Germany, March 2001.
- [15] P. Wilson. Operating System Support for Small Objects. In *Workshop on Object Orientation in Operating Systems*, pages 80–86, Palo Alto, CA, Oct. 1991. IEEE.
- [16] P. Wilson, S. Kaplan, and Y. Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proc. 1999 USENIX Tech. Conf.*, pages 101–116, Monterey, CA, June 1999.
- [17] B. Yang, R. Bryant, D. O’Hallaron, A. Biere, O. Coudert, G. Janssen, R. Ranjan, and F. Somenzi. A Performance Study of BDD-Based Model Checking. In *FMCAD’98*, pages 255–289, Palo Alto, CA, Nov. 1998.

Drive-Thru: Fast, Accurate Evaluation of Storage Power Management

Daniel Peek and Jason Flinn

Department of Electrical Engineering and Computer Science
University of Michigan

Abstract

Running traces of realistic user activity is an important step in evaluating storage power management. Unfortunately, existing methodologies that replay traces as fast as possible on a live system cannot be used to evaluate timeout-based power management policies. Other methodologies that slow down replay to preserve the recorded delays between operations are too time-consuming. We propose a hybrid approach, called Drive-Thru, that provides both accuracy and speed of evaluation by separating time-dependent and time-independent activity. We first synchronously replay file system activity on the target platform to create a base trace that captures the semantic relationship between file system activity and storage accesses. We then use the base trace as input to a simulator that can evaluate different disk, network, file cache, and file system power management policies. We use Drive-Thru to study the benefit of several recent proposals to reduce file system energy usage.

1 Introduction

The battery capacity of small, mobile computers such as handhelds limits the amount of energy that can be expended to access data. Given that I/O devices are often power-hungry, it is essential that the storage hierarchy employ power management to extend battery lifetime. Consequently, power management has been a hotspot of recent research in storage and file systems.

Unfortunately, it is often difficult for storage researchers to evaluate new power management strategies. The reason is that many of the activities associated with power management are *time-dependent*: the execution of these activities is related to the amount of time that has passed since a prior event. For example, the disk [7] and network [15] may enter power-conserving states after they have been idle for a given amount of time, file systems may coalesce operations that occur within a given time interval to save power [19] and reduce network transmissions [16], and the file cache may delay flushing dirty blocks for some time to increase I/O burstiness [28].

Methodologies that ignore this time-dependent activity are highly inaccurate, while methodologies that capture it faithfully are often too slow.

Historically, *trace replay* on a live system has been one of the most popular methods of evaluating storage systems. Using this method, one first captures representative traces of user activity, then replays the traces to measure the performance impact of proposed modifications.

Usually, trace replay on a live system omits any idle time between activities in order to speed evaluation, especially when a large set of possible configurations is being tested. While this methodology accurately captures time-independent activity, the impact of time-dependent operations cannot be measured. For example, a power management algorithm that spins down the disk during idle periods is not invoked since replay does not pause between activities.

Based on the above observation, one might reasonably decide to preserve the recorded interarrival times. In our own experience [1, 19], we have found this technique to be highly accurate; however, we have also found it to be too time-consuming. The time to run a single experiment is essentially equivalent to the length of the original trace. Potentially, one might shave some time off replay by eliminating extremely long idle periods. However, we have found that the background processing on most modern computers ensures that no more than a few seconds passes without *some* file system activity—this makes the occurrence of long idle periods rare. Given that one will often want to test multiple design options and run multiple trials to demonstrate repeatability, preserving interarrival times is too slow for all but the shortest traces.

A method to preserve interarrival times while hastening evaluation is to decrease the length and number of traces. However, it is important to ensure that the remaining traces represent realistic user activity. These microbenchmarks can be a useful guide to optimizing a storage system, but over-reliance on one or two short traces can lead to erroneous conclusions, as we show in Section 4.

A final alternative is simulation. While discrete-event simulators [5, 30] are often used to model disks, the majority of power management algorithms are implemented in other layers of the storage hierarchy (e.g. in the device driver, the file cache, and the file system). Accurate simulation of storage power management requires that the simulator capture all layers. Thus, the complexity of developing an accurate storage hierarchy simulator is a significant drawback of this approach. For instance, the implementation of these layers (i.e. the VFS layer, ext2, and the IDE driver) in the Linux 2.4 kernel comprises over 50,000 lines of source code. Even after a simulator is developed, careful testing is required to ensure that it is bug-free and that it faithfully replicates the behavior of the simulated system. Finally, substantial simulator modifications may be needed in order to evaluate new operating systems or file systems; even minor version changes to these software layers may invalidate simulation results. One might possibly use a whole system simulator [23] that can run the entire operating system within the simulator and perform trace replay on top of it. However, substantial portability issues will still arise due to the diversity of hardware platforms seen in mobile computing today; one may potentially have to develop several complex simulators to examine behavior on all target platforms.

All of the above methodologies have different strengths, but no single methodology is fast, accurate, and portable. We therefore propose a new methodology, Drive-Thru, that combines the best features of trace replay on a live system and simulation by separating time-dependent and time-independent activity. Time-independent operations, such as the mapping of file system operations to disk accesses and the determination of which requests hit in the file cache, are captured through trace replay. The output of the replay is a *base trace* that captures the semantic relationship between file system operations and disk requests. The base trace is used as input to a simulator that models only time-dependent behavior such as disk spin-down and the delayed writes of dirty cache blocks. Our validation of Drive-Thru shows that it is over 40,000 times faster than a trace replay on a live system that preserves request interarrival times. At the same time, Drive-Thru's estimates for the ext2 file system are within 5% of the measured filesystem delay and within 3% of the measured file system energy consumption. Further, since our simulator need not model complex time-independent behavior, Drive-Thru is highly portable—currently, our simulator is only 914 lines of source code.

As we discuss further in Section 8, the success of this approach rests on two assumptions: first, that we can cleanly separate time-dependent behavior from time-independent behavior, and second, that most of the com-

plexity of the storage hierarchy is time-independent.

We have used Drive-Thru to perform a detailed case study of storage power management policies. We concentrate on power management optimizations that require no application modification. For local file systems, we find that a policy that writes dirty data to disk before spin-down yields significant energy reductions, but that increasing the Linux `age_buffer` parameter beyond 30 seconds does not have a large enough benefit to offset the increased danger of data loss. We also note that operating systems should be careful not to set `age_buffer` to a value that is close to the spin-down time of the hard drive. We find substantially different behavior for a network file system. The policy of writing dirty data before using a network power saving mode *increases* energy usage. Further, writes need be delayed only 2 seconds for energy-efficiency; almost no additional energy reduction is realized by delaying writes further.

We begin with a description of the design and implementation of Drive-Thru. Section 3 evaluates the speed and accuracy of our methodology. Section 4 and Section 5 present case studies of power management strategies for a local and a network file system, respectively. Section 6 applies the results of these case studies to optimize an existing file system. After discussing related work and Drive-Thru's limitations, we conclude.

2 Drive-Thru: design and implementation

2.1 Overview

Figure 1 shows an overview of the Drive-Thru methodology. We begin with a pre-recorded trace of file system activity that captures POSIX operations such as `mkdir`, `open`, and `unlink`, as well as the time at which each operation occurred. Such traces may be found in the public domain [16]; alternatively, we have built a trace capture tool that allows us to record our own.

It is important to note that block-level traces that record disk accesses are insufficient for our purpose. A block-level trace does not capture enough semantic information to reason about how power management at higher layers of the system will affect performance and energy usage. Each disk trace captures one particular file system and cache behavior, so considering alternative behaviors is infeasible. For instance, if dirty blocks can reside in the file cache for up to 30 seconds, then two writes to the same block 20 seconds apart are coalesced into a single disk write. By examining only the disk trace, one cannot determine that two writes rather than one would occur if the write-back delay is reduced to 15 seconds.

As described in Section 2.2, we first replay each file system trace on the target platform, operating system, and

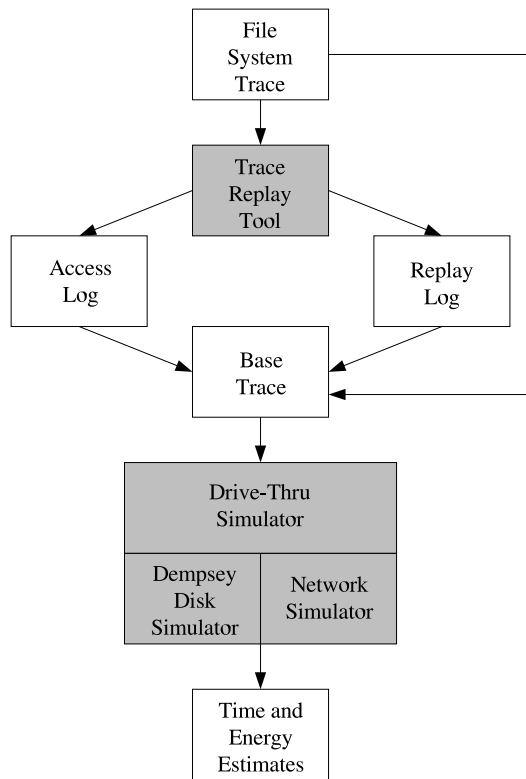


Figure 1. Drive-Thru overview

file system. The trace replay tool records the time when each file system operation begins and ends in the *replay log*. Simultaneously, we also generate a low-level *access log* that records each I/O access during the trace. After each POSIX operation completes, the trace replay tool ensures that all modifications have been reflected to storage (e.g. by using the `sync` system call) and then immediately begins the next operation. Reflecting modifications to storage immediately allows us to associate each I/O access with the POSIX operation that caused it. Because we omit recorded idle time, the base trace is generated in much less time than it took to record the original trace.

The next step, described in Section 2.3 is to merge the original trace, the access log, and the replay log into a *base trace* that captures the semantic relationship between high-level file system operations and low-level accesses. This relationship reveals time-independent behavior such as how a particular file system lays out its data on disk and which accesses hit in the file cache. The base trace is input to the Drive-Thru simulator, which models time-dependent behavior.

Drive-Thru simulates time-dependent activity at each layer of the hierarchy. For example, at the file system

layer, it models queuing behavior in distributed file systems such as Coda [14] and BlueFS [19] where file operations are delayed before being sent over the network to the file server. At the file cache layer, the simulator models the coalescing of block writes due to delayed writeback of dirty blocks. At the device driver layer, the simulator models specific power management algorithms such as disk spin-down [7] or the use of 802.11b's Power Saving Mode (PSM) [11].

As described in Section 2.4, the Drive-Thru simulator delays, eliminates, and coalesces I/O accesses depending upon the specific power-management algorithms specified at each layer of the storage hierarchy. Each resulting I/O access is then passed to a device-specific simulator that models network or storage. For disk simulation, we use Dempsey [30], a modified version of DiskSim [5] that simulates both time and energy. For network simulation, we use a custom 802.11b simulation module. As output, Drive-Thru gives the expected time and energy to replay the trace on the target platform. By changing the parameters of the Drive-Thru simulator, one can quickly investigate alternative file system, cache, and device power management strategies.

2.2 Replay

The Drive-Thru file system trace replay tool runs on the target platform for evaluation. It replays operations recorded in the trace sequentially and forces all dirty data to storage before beginning each new operation. Since all file system operations issued by the replay tool are POSIX-compliant, the replay tool is portable; i.e., it can be run on any POSIX platform.

During replay, we record disk accesses using a modified `ide-disk` Linux kernel module. The module generates a disk access log that records the start time, completion time, starting sector, length, and type (read or write) of every access. The module writes this data to a 1 MB kernel buffer that is read by a user-level program through a pseudo-device after the trace is complete.

We employ a similar strategy to monitor network accesses made by distributed file systems. We modify the file system's remote procedure call (RPC) package to record the start time, completion time, size, and type (read or write) of each RPC. This information is written to a network access log—on the iPAQ platform, this log is stored in RAM.

2.3 Generating the base trace

As shown in Figure 2, we merge the original file system trace, the replay log, and the access log(s) to generate a *base trace*. The base trace preserves the interarrival times between file system operations that were captured in the

File System Trace	Replay Log	Access Log	Base Trace
Stat (file1)	Stat (file1)	Read (sector 16, length 8)	Stat (file1) Read (sector 16, length 8)
Sleep (5 s.)			Sleep (5 s.)
Chmod (file2)	Chmod (file2)	Write (sector 80, length 8)	Chmod (file2) Write (sector 80, length 8)
Open (file3)	Open (file3)	Read (sector 96, length 8) Read (sector 48, length 8)	Open (file3) Read (sector 96, length 8) Read (sector 48, length 8)
Stat (file1)	Stat (file1)		Stat (file1)

Figure 2. Example of generating a base trace

original file system trace—this is shown by the `sleep` record in the trace in Figure 2.

For each file system operation, the base trace shows the disk and/or network accesses generated by executing each operation on the target platform. Because the trace replay tool executes each operation sequentially and ensures that all dirty data is flushed before it begins the next operation, each storage access will overlap with exactly one file system operation (although a single file system operation may overlap several storage accesses). This methodology makes clear the semantic relationship between file system operations and disk and network accesses.

As shown in Figure 2, the base trace captures the relationship between the file system trace and the access trace—this shows time-independent behavior such as how the file system lays out its data on disk and which blocks hit in the file cache. Because the base trace already captures this behavior, the Drive-Thru simulator need only simulate time-dependent operations.

2.4 The Drive-Thru simulator

Using the base trace as input, the Drive-Thru simulator calculates the time and energy that it would take to run the file system trace on the target platform had interarrival times been preserved during trace replay. Drive-Thru uses discrete-event simulation to model three layers of the storage hierarchy: the file system, the file cache, and device power management. At each layer, the simulator modifies the stream of I/O accesses seen in the base trace. Based upon the time-dependent behavior specified, the simulator:

- *Delays I/O accesses.* Disk or network accesses are often delayed to save power. For example, on the Hitachi 1 GB microdrive, once the drive enters its standby mode, the next access is delayed approximately 800 ms [10]. Similarly, a transition between power modes on a 802.11b network interface delays accesses for several hundred milliseconds until the transition completes [1]. At the file system and cache layers, write accesses are often delayed to improve performance or save power. Based on the behavior specified, the Drive-Thru simulator may delay accesses at each layer of the storage hierarchy. This can result in reordering of accesses (e.g. if writes are delayed but reads are not).
- *Eliminates I/O accesses.* A delayed access may be obviated entirely by a subsequent file system operation, in which case the simulator eliminates it. For example, the `age_buffer` parameter specifies how long a dirty block may remain in the file cache until it is written to storage. If a write operation creates a dirty block that is overwritten by a subsequent write operation within `age_buffer` seconds, only one disk access will result. In this case, the simulator eliminates the first disk write upon seeing the second write. Similarly, the Coda file system delays sending operations to the file server for up to 300 seconds in write-disconnected mode [16]. If two operations that cancel each other are performed within this period, neither is sent over the network. The simulator would therefore eliminate all network accesses associated with the two operations.

- *Coalesces I/O accesses.* Two discrete I/O accesses may be merged into a single large access. For example, two writes to adjacent disk sectors can be merged into a single write to two sectors. Similarly, the Blue file system aggregates RPCs that modify data in order to save power [19]. Thus, for BlueFS, our simulator coalesces multiple small network accesses into a larger access whose size is the sum of the sizes of the smaller RPCs.

After the Drive-Thru simulator accounts for time-dependent behavior in the file system, file cache, and device power management layers, it hands the resulting accesses off to a device-specific simulation module to model the time and energy needed to perform the network and disk accesses. For disk simulation, we use the Dempsey simulator [30] developed by Zedlewski et al. For network simulation, we use our own custom module that models 802.11b behavior.

Dempsey is a discrete-event simulator based on DiskSim [5] that estimates the time and energy needed to perform disk accesses. Dempsey calculates when each disk access will complete based upon a detailed characterization of the drive. At the end of simulation, Dempsey reports the total energy used by the drive. Drive-Thru adds this value to its calculated energy usage for the network and the rest of the system to derive the total energy usage of the mobile computer.

The network simulator takes as input the measured latency and bandwidth between the mobile computer and file server, as well as a model of packet loss. The network simulator currently models three power management strategies: the Continuous Access Mode (CAM) and Power-Saving Mode (PSM) defined by the 802.11b standard [11], as well as Self-Tuning Power Management (STPM) [1], an adaptive strategy that toggles between CAM and PSM depending upon the observed network access patterns. The network module calculates the time and energy used to perform each RPC based upon a characterization of the network card that includes the power needed to transmit and receive data in each mode as well as the time and energy cost of switching between CAM and PSM. At the end of simulation, the module reports the total energy used by the network.

3 Validation

We validated Drive-Thru by comparing its time and energy results to those obtained through a trace replay on a live system that preserves request interarrival times.

3.1 Methodology

We evaluated Drive-Thru on an HP iPAQ 3870 handheld running the Linux 2.4.19-rmk6 kernel. The handheld

has a 206 MHz StrongArm processor, 64 MB of DRAM and 32 MB of flash. We used a 1 GB Hitachi microdrive [10] and a 11 Mb/s Cisco 350 802.11b network card in our experiments. The Hitachi microdrive uses the adaptive ABLE-3 controller that spins down the disk to save power. Empirical observation of its behavior reveals that the microdrive spins down once it has been idle for approximately 2 seconds.

We measured operation times using the `gettimeofday` system call. Energy usage was measured by attaching the iPAQ to an Agilent 34401A digital multimeter. We removed all batteries from the handheld and sampled power drawn through its external power supply approximately 50 times per second. We calculated system power by multiplying each current sample by the mean voltage drawn by the mobile computer — separate voltage samples are not necessary since the variation in voltage drawn through the external power supply is very small. We calculated total energy usage by multiplying the average power drawn during trace replay by the time needed to complete the trace. The base power of the iPAQ when idle with no network or disk card inserted is 1.4 Watts. If we had turned off the screen, the iPAQ would have used only 0.9 Watts.

We used file system traces from two different sources. The first four in Figure 3 were collected by Mummert et al. at Carnegie Mellon University [17] from 1991 to 1993. The remaining traces are NFS network traces collected by Kim et al. at the University of Michigan in 2002 [13]. Because it would be too time consuming to replay each trace in its entirety, we selected segments approximately 45 minutes in length from a subset of these traces.

We first replayed each segment on the iPAQ, preserving operation interarrival times and measuring the time and energy used to complete each trace. We then compared the measured results with Drive-Thru's estimates. However, we note that the majority of time spent during trace replay is due to recreation of interarrival time rather than file system activity. A reasonable metric of accuracy may therefore be how well Drive-Thru estimates the non-idle time in the trace. We calculate this *file system time* by subtracting the idle time from the total trace time; this metric reflects the additional delay added by the file system. We also calculate *file system energy*, which is the amount of additional energy consumed by file system activity during trace replay. To generate this value, we subtract the product of the handheld's idle power and the idle time from the total energy used to replay the trace.

3.2 Local file system results

We began by evaluating how accurately Drive-Thru estimates the time and energy needed to replay traces on

Trace	Number of Ops.	Length (Hours)	Update Ops.	Working Set (MB)
Purcell	87739	27.66	6%	252
Messiaen	44027	21.27	2%	227
Robin	37504	15.46	7%	85
Berlioz	17917	7.85	8%	57
NFS2	39074	24.00	28%	21
NFS15	34188	24.00	16%	4

This figure shows the file system traces used in our evaluation. Update operations are those that modify data. The working set is the total amount of data accessed during a trace.

Figure 3. File traces used in evaluation

Linux’s ext2 file system. Figures 4(a) and 4(b) compare the measured time and energy to replay traces on ext2 with Drive-Thru’s estimates. For the Purcell trace, we vary Linux’s file cache write-back delay (*age_buffer*) from 15 to 60 seconds; all other segments were replayed with the default 30 second value. As can be seen, Drive-Thru’s estimates of total replay time and energy are extremely accurate—on average, they are within 0.10% of the measured time to execute each trace and within 0.21% of the measured energy.

As shown in Figures 4(c) and 4(d), Drive-Thru still maintains excellent accuracy when only file system time and energy are considered. Drive-Thru’s estimates are, on average, within 5% of the measured file system time and within 3% of the measured file system energy usage.

3.3 Network file system results

We next validated Drive-Thru using a network file system in which data is stored on a remote server rather than local disk. For this purpose, we used the Blue file system [19], a distributed file system developed at the University of Michigan. One reason that we chose BlueFS for this study is that it has been designed from the start with energy-efficiency as a goal; consequently, BlueFS has been shown to use much less energy than other distributed file systems. BlueFS stores the primary replica of each file system object on a network file server. Further, it can optionally cache second-class replicas on local and portable storage devices. However, for this particular case study, we allow BlueFS to use only the remote file server so as to isolate the performance of Drive-Thru for network storage.

BlueFS issues remote procedure calls (RPCs) to the network server in a manner similar to NFS [18], e.g. each RPC roughly corresponds to an individual VFS operation. BlueFS queues RPCs that modify data for up to 30 seconds before sending them to the server. On the iPAQ, BlueFS also flushes the outgoing queue if the size of queued data exceeds 11.25 MB. If the queue size exceeds 15 MB, new file system operations are blocked until some operations are sent to the server. When send-

ing queued operations to the server, BlueFS coalesces RPCs into aggregate RPCs of up to 1 MB. We modeled this time-dependent behavior in the Drive-Thru simulator with 44 lines of source code.

In Figures 4(e–h), we show the accuracy of Drive-Thru’s estimates for the Purcell and NFS15 traces. For each trace, we examine behavior using 802.11b’s CAM and PSM modes [11], as well as the adaptive STPM policy [1]. On average, Drive-Thru’s estimates are within 0.86% of measured values for total time and within 0.90% for total energy. When we remove idle time from consideration, Drive-Thru’s estimates are, on average, within 13% for file system time and within 7% for file system energy usage.

Why is Drive-Thru less accurate for network file systems than for local disk file systems? Partially, this is because network accesses are inherently more variable (as can be seen by the larger error bars in the measured results for network accesses in Figure 4). Also, we do not yet model the storage hierarchy of the network file server. Finally, we might be able to improve Drive-Thru’s estimates by replacing our simple network simulator with one that is more accurate [12, 26].

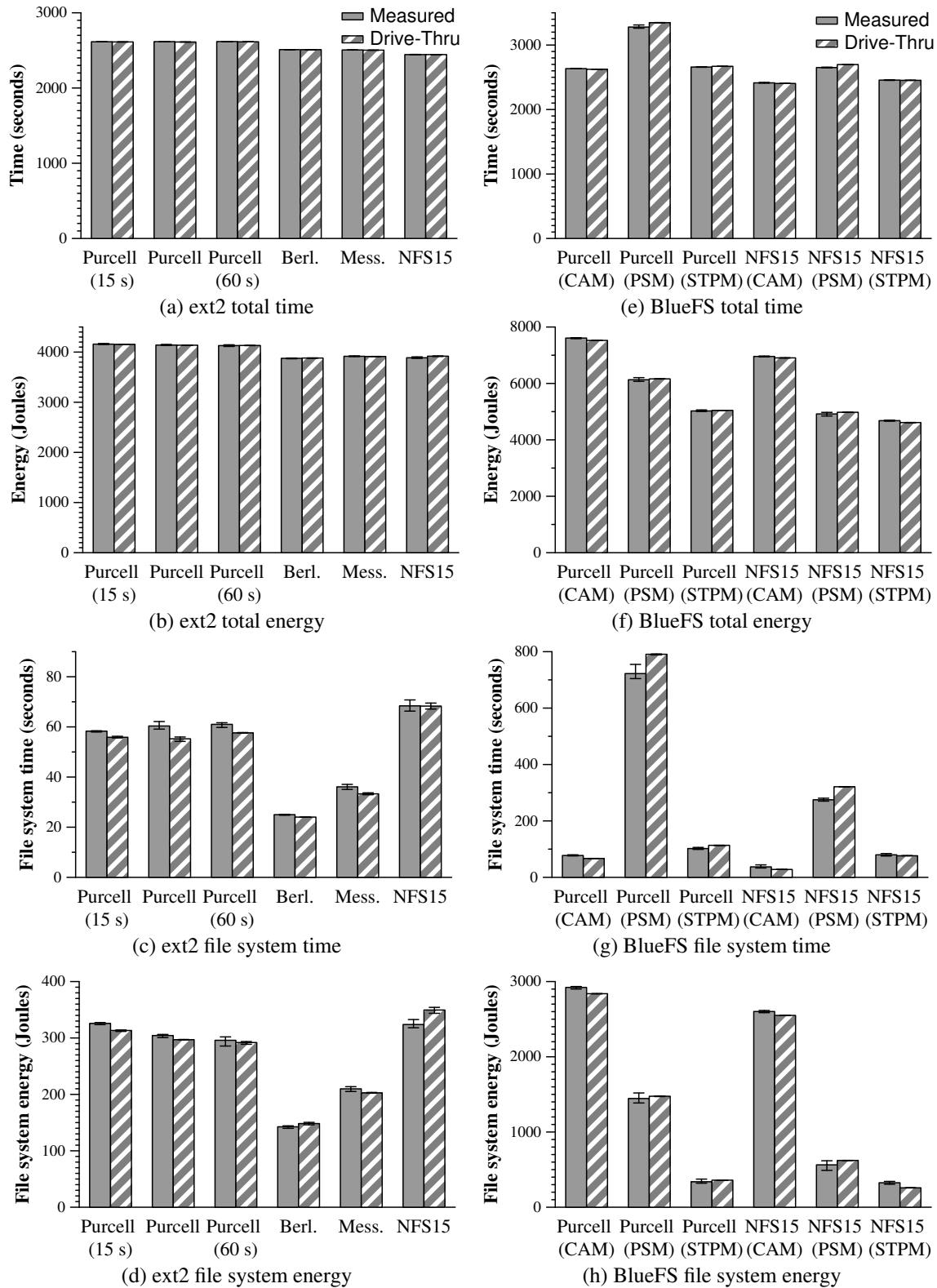
4 Case study: local file system

4.1 Methodology

Using Drive-Thru, we quantified the benefit of several recent proposals that modify file cache behavior in order to reduce energy consumption. We selected for our study only proposed modifications that could easily be implemented in current commodity operating systems. As a consequence, none of the proposals we selected require changes to existing application source code.

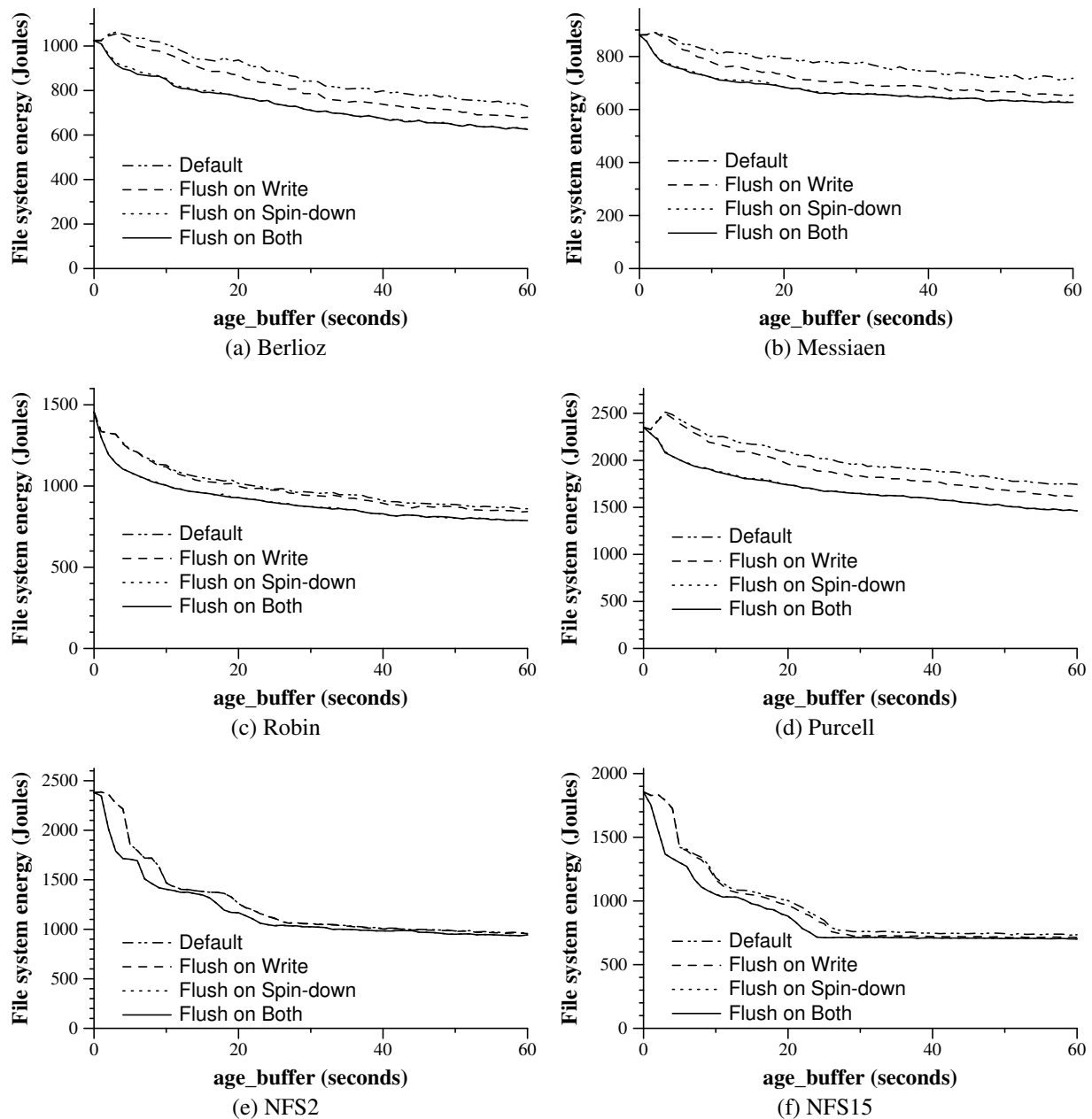
The three modifications that we studied were:

- *Flush on write.* Whenever any single dirty cache block is written to disk, write all dirty cache blocks. This modification was proposed by Weissel et al. [28], who referred to it as the “write back all buffers” strategy. Papathanasiou and Scott also propose to write all dirty blocks in their energy-efficient prefetching work [21].
- *Flush on spin-down.* Before the disk is spun down to save power, write all dirty blocks in the cache to disk. Weissel [28] first proposed this strategy, referring to it as “Update on shutdown”.
- *Increasing *age_buffer*.* This allows a dirty block to reside in the cache for a longer period of time. The default Linux *age_buffer* value is 30 seconds. Weissel et al. [28] increased this value to



The left column validates Drive-Thru for a local file system by comparing its time and energy estimates with measured results for ext2. The right column validates a network file system by comparing results for BlueFS. File system time and energy are metrics that reflect the overhead of the file system. Each bar is the mean of three trials, with error bars giving the value of the lowest and highest trial. Note that the scales are different between the two columns due to the larger overhead of network file systems.

Figure 4. Drive-Thru validation



Each graph shows the results of implementing the three cache management policies described in Section 4.1 for the ext2 file system. We show only energy results since the policies studied had negligible performance impact. Note that the scales are different in each graph. In most graphs, the "Flush on Spin-down" and "Flush on Both" lines overlap and thus appear to be only a single line. Similarly, the "Default" and "Flush on Write" lines overlap in some graphs such as 5(e).

Figure 5. Results from local file system case study

60 seconds in their work. Papathanasiou [21, 22] also increased this value to 60 seconds (or longer in some cases). In our own work [1], we have proposed a similar strategy that delays writes as long as possible for data that is replicated elsewhere.

We used Drive-Thru to assess the potential benefit of each strategy for the ext2 file system on the iPAQ de-

scribed in Section 3. We used the six full-length file traces shown in Figure 3. We examined the four combinations of enabling or disabling *flush on spin-down* and *flush on write*, while varying *age_buffer* from 0–60 seconds. Note that *age_buffer* only controls the eligibility of dirty blocks to be written from the Linux cache; the updated thread that actually writes the data wakes up

every interval seconds. By default, `age_buffer` is set to 30 seconds and `interval` is set to 5 seconds (meaning that a dirty block may actually reside in the cache for up to 35 seconds before flushing starts). As we varied `age_buffer`, we also varied `interval` to preserve the default 6:1 ratio.

4.2 Results

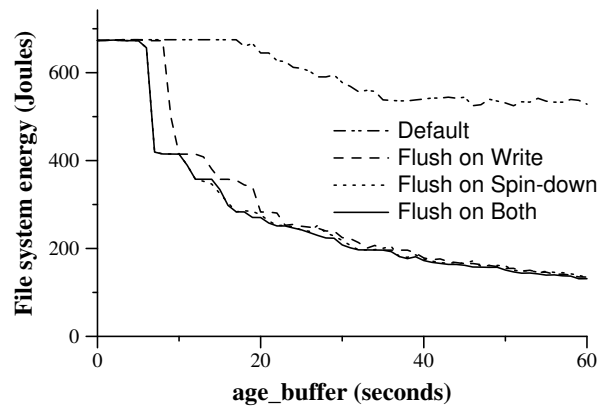
As can be seen in Figure 5, both the *flush on spin-down* and the *flush on write* policies are effective in reducing energy usage for almost every trace. However, the *flush on write* policy is less effective; in the NFS2 trace, for example, its energy usage is essentially equivalent to the default Linux cache management policy. Interestingly, for all six traces, the *flush on spin-down* policy alone saves almost the same amount of energy as the combination of the two policies. Examination of the detailed results shows that if the cache is flushed on spin-down, the *flush on write* mechanism is almost never used as long as the value of `age_buffer` is larger than a few seconds. Essentially, it is rare for read operations to constantly keep the disk spinning long enough for the *flush on write* policy to take effect.

Across all six traces, *flush on spin-down* reduces file system energy usage by an average of 11%. Since *flush on spin-down* is relatively easy to implement and because it requires no application modification to be effective, we recommend that the policy be added to commodity operating systems such as Linux.

We were surprised to see that increasing `age_buffer` beyond the Linux default of 30 seconds has little effect on energy consumption. On average, increasing `age_buffer` from 30 to 60 seconds reduces file system energy usage by only 8% (assuming *flush on spin-down* is implemented). Increasing the value to more than 60 seconds does not significantly decrease energy usage further. Intuitively, the effect is minimal because the distribution of disk accesses is such that few interarrival times fall between 30 and 60 seconds. Unlike the other two policies, increasing `age_buffer` has a substantial downside because it increases the window during which data may be lost due to system crash. Therefore, we recommend that `age_buffer` be left unchanged unless storage energy usage is an overriding concern.

Close examination of the trace results revealed a feature that we did not predict. In Figure 5(d), the default Linux cache policy experiences a peak in energy consumption when `age_buffer` is set to 2 seconds. This peak exists in all other traces, although it is not particularly pronounced in some. In addition to using Drive-Thru, we have also confirmed the existence of this peak experimentally.

By studying Drive-Thru trace results, we determined that this peak occurs when `age_buffer` is set to a value



This shows the results of implementing the three cache management policies described in Section 4.1 on the energy needed to download a 37 MB file and store it on disk.

Figure 6. Results from download microbenchmark

that closely approximates the disk spin-down timeout. In all of our traces, reads and writes tend to be clustered together due to the bursty nature of file system accesses [27]. For each cluster, the disk spins up to service the read requests, then spins down immediately before updated writes back dirty cache blocks modified by operations in the cluster. The disk must then spin up to service updated writes; it also remains in a high-power state for another two seconds before spinning down again. Note that use of the *flush on spin-down* avoids this peak because it writes dirty blocks to the cache before the first spin-down operation. To avoid this behavior, we recommend that the operating system set the disk spin-down threshold to a value different from `age_buffer` (or else, implement *flush on spin-down*).

Calculating time and energy estimates using Drive-Thru requires two steps: the generation of the base trace, followed by the execution of the simulator. When comparing the speed of Drive-Thru to traditional trace replay, the worst-case comparison is when Drive-Thru is only used to generate a single data point. For the six traces that we studied, Drive-Thru was 158–614 times faster than trace replay for this task. Further, for the entire case study, we only needed to generate the base trace once for each file system trace. This means that the effort to generate the base trace is amortized over many runs of the simulator. In fact, if we were to generate the results in this study using trace replay, it would have taken over 40,000 times longer than the time required using Drive-Thru. Put in raw terms, we can generate all the results in this case study in less than 45 minutes using Drive-Thru; the same results would take over 3 years to generate using trace replay on a live system.

4.3 The danger of microbenchmarks

Seemingly, our results for ext2 contradict previous results reported in the literature that show greater benefit for these cache management strategies. We believe that both measurements are accurate, but that the difference between the two results shows the danger of over-reliance on microbenchmarks. We tried to confirm this by replicating one experiment [8] where data is downloaded to a mobile computer and stored on local disk. In our experiment, we download a 37 MB file over the iPAQ serial port and write it to the microdrive. The download rate over the serial link is approximately 8 KB/s.

We first generated a base trace by running the download application. We then used Drive-Thru to produce the results shown in Figure 6. Note that both *flush on spin-down* and *flush on write* produce substantially larger energy savings for this microbenchmark (61% and 62% respectively). Once these policies are implemented, increasing `age_buffer` from 30 to 60 seconds yields a 37% reduction in energy usage.

Although these results are substantially more impressive than those that are generated when we apply the policies to our file traces, we believe that the two sets of results are consistent. Some applications captured in the file system trace may well have access patterns that yield benefits similar to the download microbenchmark. However, there may be many other applications using the file system for which the policies under study are considerably less effective.

We also believe that these results demonstrate the primary benefit of Drive-Thru: because evaluation is dramatically faster using our methodology, it is possible to evaluate new storage power management policies on a much wider set of data. As the scope of data used for validation increases, the danger that results reflect a particular feature of the workload is diminished.

5 Case study: network file system

5.1 Methodology

We next examined how well the policies studied in the previous section translate to a network file system. In this study, we use BlueFS, described previously in Section 3.3. As before, we do not allow BlueFS to use local storage so as to isolate the effect of network storage.

Recall that 802.11b defines a high-performance Continuous Access Mode (CAM) and a low-power Power-Saving Mode (PSM). The STPM policy used by BlueFS adaptively switches between these two modes depending upon observed network traffic [1]. In these experiments,

it switches to CAM before beginning the third consecutive foreground RPC received within a short time window, and it switches to PSM after the network interface is idle for 300 ms. In the context of BlueFS and STPM, we therefore define the three policies studied as:

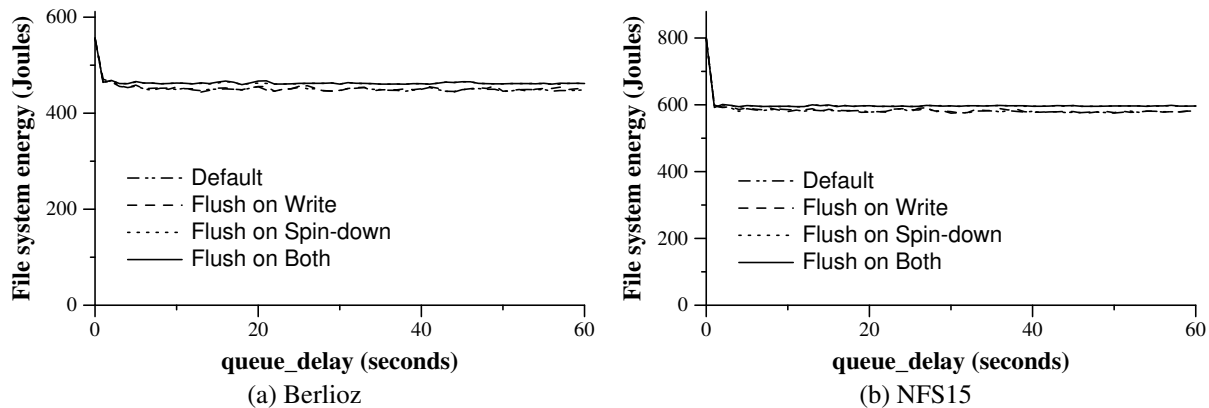
- *Flush on write*. Whenever any single RPC containing a modification is sent to the server, all queued modifications are also sent to the server.
- *Flush on PSM*. Before the network card is placed in PSM to save power, all queued modifications are sent to the server. This is the network equivalent of *flush on spin-down*.
- *Increasing queue_delay*. This allows a modification to be queued by the file system for a longer period of time. This is the BlueFS equivalent of `age_buffer`. By default, BlueFS sets `queue_delay` equivalent to Linux's default `age_buffer` of 30 seconds.

We used Drive-Thru to assess the potential benefit of each strategy for BlueFS running on the iPAQ described in Section 3. We again used the six full-length file traces shown in Figure 3. However, since the results of the study for all six traces were essentially equivalent, we show results for only two traces in Figure 7.

5.2 Results

Comparing Figures 5 and 7, it is immediately apparent that the three policies under study have a markedly different effect on the network file system than they do on the local disk file system. The *flush on PSM* policy appears to have a slight negative effect, increasing file system energy usage for all values of `queue_delay` greater than a few seconds. The *flush on write* policy has no noticeable impact, using essentially the same energy as the default policy. We have also observed that neither policy has a significant effect on trace replay time.

We believe that the observed difference in effectiveness is due to the difference between disk and network power management. While a disk is spun-down to save power, no data may be read from the drive; it must therefore spin up to service any write request that it receives. In contrast, when the network interface is in PSM, it can still transmit and receive data. While the latency of each RPC increases in PSM, the energy usage does not. Since write RPCs are asynchronous (i.e. they are sent by a background thread), STPM can (and does) leave the network interface in PSM while they are sent and received. Because write RPCs can efficiently be serviced in PSM, it is not necessary to flush them from the queue before transitioning the network card.



Each graph shows the results of implementing the three cache management policies described in Section 5.1 for BlueFS. We show only energy results since the policies studied had negligible performance impact. Note that the scales are different in each graph.

Figure 7. Results from network file system case study

The effect of `queue_delay` in Figure 7 is also striking. While all six traces show considerable benefit from setting `queue_delay` greater than zero, no trace showed more than a trifling benefit from increasing `queue_delay` beyond two seconds. This came as a complete surprise to us (the creators of BlueFS), since we expected the effect of `queue_delay` to mirror that of `age_buffer` for `ext2`.

In BlueFS, `queue_delay` directly affects the durability of file data, since writes are only guaranteed to be safe from client crash once the server replies to a RPC. Since there is almost no performance or energy cost, this study made a convincing case to us that we should decrease `queue_delay` from 30 to 2 seconds for the BlueFS server write queue—we explore this further in the next section.

The speedup obtained by using Drive-Thru is somewhat less for network file systems than for local file systems. The base trace takes longer to generate since operations must be performed synchronously and each operation requires a network round-trip. However, even for the generation of a single data point, Drive-Thru was 38–116 times faster than trace replay for the file system traces we studied. In total, we were able to generate the results in this case study over 13,000 times faster using Drive-Thru than we would have been able to generate them if we had used trace replay on a live system.

6 Optimizing the Blue file system

We are applying the lessons from these case studies to BlueFS. Since BlueFS layers its client disk cache on top of `ext2`, the results of the first case study apply to its local cache, whereas the results of the second case study apply to its network communication.

The first lesson is that different storage devices require different policies. Policies such as *flush on spin-down* that benefit disk storage are detrimental to network storage. We therefore changed BlueFS to allow these policies to be selectively applied to each network or local storage device attached to the mobile computer. Note that BlueFS already allowed each network or local storage device to select its own `queue_delay`.

Our second lesson is that we should implement *flush on spin-down* for local disk-based storage. We modified the disk power management module used by BlueFS to call the Linux kernel routine that flushes dirty cache blocks for a particular device (`fsync_dev`) before the module spins down the disk. This required only 20 lines of source code.

We ran the 45-minute Purcell trace used in Section 3 on BlueFS with and without our modification. Our experimental setup was the same as before, except that we allowed BlueFS to use both the network server and a local cache on the microdrive’s `ext2` file system. As shown in Figure 8, using *flush on spin-down* for the local cache reduces file system energy usage by 12.4%. Note that this improvement is for a file system that has already been substantially optimized to reduce energy usage. Given that this change also reduces trace execution time, it seems a nice improvement. Although total energy savings for the entire system is only 1.4%, our experiment assumes that the handheld does not hibernate or employ similar system-wide power-saving modes during periods of inactivity. We make this conservative assumption because our file system traces do not record sufficient information to predict when the system would hibernate.

Finally, we decreased the value of `queue_delay` for the server write queue from 30 seconds to 2 seconds. Using the same 45-minute Purcell trace, we evaluated the

	Original BlueFS	BlueFS with flush on spin-down	Percentage improvement
Time (s)	2674 (2671–2677)	2663 (2661–2665)	0.4%
Total energy (J)	5482 (5465–5506)	5408 (5394–5423)	1.4%
File system time (s)	118 (115–121)	105 (105–107)	11.0%
File system energy (J)	597 (580–621)	523 (509–537)	12.4%

This shows the results of implementing the flush on spin-down policy in BlueFS. Each value reports measured replay time or energy usage for an approximately 45 minute segment of the Purcell file system trace. Each value shows the mean of three trials with the minimum and maximum given in parentheses.

Figure 8. Flush on spin-down in BlueFS

behavior of these two settings for the network-only configuration of BlueFS (omitting the local disk cache). Our results showed that the difference in energy usage between these two values was within experimental error of the 6 Joule difference predicted by Drive-Thru. Further, the time to replay the trace was equivalent within experimental error for the two settings. The benefit of less data loss after a crash seems well worth this cost.

7 Related Work

To the best of our knowledge, Drive-Thru is the first system to quickly and accurately evaluate storage power management by separating out time-dependent and time-independent behavior. The foundations of our methodology lie in trace replay, which has long been used as an evaluation technique in the storage research community [5, 16, 20, 29]. Recent efforts to improve the accuracy of trace replay [3] could benefit our methodology. However, our key observation is that trace replay that does not preserve operation interarrival times is too inaccurate to model storage power management, while trace replay that preserves interarrival times is too slow. Thus, while trace replay has been used previously to evaluate storage power management [1, 2, 19], the traces used in those studies have been, of necessity, quite short.

Drive-Thru’s process of generating the base trace can be viewed as instance of a gray-box system [4]. Both Drive-Thru and the gray-box File Cache Content Detector (FCCD) use a real file cache to estimate cache contents. In contrast to our work, FCCD provides on-line estimates of cache contents to applications. Drive-Thru’s cache content prediction is fundamentally off-line. Our current approach for evaluating each storage hierarchy is to modify the Drive-Thru simulator parameters in order to reflect the file system, cache, and device power management strategies for the system under study. Potentially, it may be feasible to automatically extract Drive-Thru parameters using techniques such as those outlined for semantically-smart disk systems [25].

Discrete event simulation of disk [5] and network [12, 26] behavior is not a contribution of this work. In fact, we

use the Dempsey disk simulator directly in our methodology. Similarly, we could potentially use a network simulator such as QualNet [26] or NS-2 [12] to improve Drive-Thru’s estimates for network file systems. The focus of our work is above the physical device in the storage hierarchy. We concentrate on the file system, file cache, and device driver where storage power management algorithms are typically implemented.

Similarly, all of the power management strategies that we investigate in our case study have been previously proposed by the research community. The contribution of our work is that we perform the first systematic study of these proposals that uses substantial traces of actual user activity to measure impact on typical file system workloads. Given the several surprising results that we generated through our case study, we believe that this type of comprehensive evaluation is of considerable merit.

A very large number of fixed and adaptive strategies have been proposed for controlling the power state of hard disks [7, 6, 9, 24]. This decision can be augmented by considering additional information provided by applications [2, 8, 28] or based on an OS-level energy budget [31]. It has also been observed that since disk power management algorithms are limited by the lengths of continuous idle time, it might be beneficial to rearrange or prevent disk accesses so that idle times could become longer [8, 22, 28].

8 Discussion and Future Work

We believe that the results of our case studies demonstrate the effectiveness of Drive-Thru’s approach to power management evaluation. Because Drive-Thru generates accurate results quickly, we were able to look at a much larger set of data than previous researchers have used in their evaluations. This led to a number of surprising results, three of which we implemented in the Blue file system that we are currently developing.

At the same time, Drive-Thru has several limitations that must be acknowledged. First, our methodology relies on the ability to separate time-dependent and time-

independent behavior in storage hierarchies. This separation is not always clear. For instance, dirty blocks may be prematurely evicted from the Linux file cache due to memory pressure. Since the trace replay tool generates little memory pressure of its own, we have never seen this happen. However, Drive-Thru may not be able to handle extremely write-intensive workloads that write blocks at a faster rate than updated writes them to disk. We note that even on an iPAQ with an extremely limited memory capacity (64 MB), our results remain quite accurate.

A second limitation of Drive-Thru is that the accuracy of its evaluation results depend upon how well the particular traces used in the evaluation reflect the actual behavior of the system under study. Good traces of file system activity are a rare commodity. Thus, it may be hard to find a precisely representative sample. Certainly, the traces we used in our study may not be representative of the workload one might see on a Linux handheld. However, we note that this is a limitation of trace replay in general, and is not specific to Drive-Thru. Further, we believe that file system traces, even those not specifically collected on a target platform, are often more representative than application microbenchmarks. In the future, we plan to help remedy this limitation by using the trace capture tool that we have developed to generate traces of file system activity from handhelds and other small, mobile devices.

A final limitation of the Drive-Thru methodology is that we assume that the time-dependent component of storage system behavior is easy to model. This assumption has held for all the systems that we have modeled so far; we have been able to specify time-dependent behavior in a few hundred lines of source code or less. However, as storage systems continue to evolve and become more complex, it is possible that this assumption will no longer hold. One remedy, mentioned in the previous section, may be to use semantically-smart techniques to automatically extract simulation parameters.

9 Conclusion

We believe that the speed, accuracy, and portability of Drive-Thru make it an extremely valuable tool for the evaluation of storage power management policies. For our study of ext2, we were able to generate results over 40,000 times faster than trace replay on a live system that preserves operation interarrival times. At the same time, our results are within 3-5% of the values that trace replay generates. Further, since our trace replay tool is POSIX-compliant and our simulator is less than 1,000 lines of source code, we believe that Drive-Thru will port relatively easily to new file systems and platforms.

The case studies reported in this paper show the value of our tool. By studying a more comprehensive set of work-

loads than those studied in the past, we have been able to gain the following insights about the effect of cache behavior on storage power management:

- For the ext2 file system, a *flush on spin-down* policy reduces file system energy usage by 11% with no performance cost and without the need to modify applications. Further, if *flush on spin-down* is implemented, a *flush on write* policy realizes little additional benefit.
- Conversely, for the BlueFS network file system, *flush on spin-down* increases energy usage, while *flush on write* has no significant effect.
- Increasing the Linux `age_buffer` parameter to a value greater than 30 seconds yields little energy savings (8%) to offset the additional danger of data loss during system crash.
- The BlueFS `queue_delay` parameter for network RPCs can be reduced to 2 seconds to improve consistency without substantially sacrificing performance or energy usage.
- Operating systems should not set `age_buffer` and the disk spin-down time to the same value.

The last two results came as complete surprises to us. This demonstrates an additional benefit of Drive-Thru: because we are able to evaluate a large parameter space quickly, we have the opportunity to mine for insights that would not otherwise be apparent. We are currently using Drive-Thru in our ongoing development of the Blue file system. Our efforts have already yielded the three enhancements described in Section 6. Based on these results, we are confident that Drive-Thru can be of considerable use in developing energy-efficient storage.

Acknowledgments

We thank Fengzhou Zheng and the Dempsey team for making their tool available to us. We would also like to thank Minkyong Kim, Jay Lorch, and Lily Mummert for providing file system traces. Manish Anand, Edmund B. Nightingale, Ya-Yunn Su, the anonymous reviewers, and our shepherd, Carla Ellis, made many suggestions that improved the quality of this paper. The work is supported by the National Science Foundation under award CCR-0306251, and by an equipment grant from Intel Corporation. Jason Flinn is supported by NSF CAREER award CNS-0346686. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, the University of Michigan, or the U.S. government.

References

- [1] ANAND, M., NIGHTINGALE, E. B., AND FLINN, J. Self-tuning wireless network power management. In *Proceedings of the 9th Annual Conference on Mobile Computing and Networking* (San Diego, CA, September 2003), pp. 176–189.
- [2] ANAND, M., NIGHTINGALE, E. B., AND FLINN, J. Ghosts in the machine: Interfaces for better power management. In *Proceedings of the 2nd Annual Conference on Mobile Computing*

- Systems, Applications and Services* (Boston, MA, June 2004), pp. 23–35.
- [3] ANDERSON, E., KALLAHALLA, M., UYSAL, M., AND SWAMINATHAN, R. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the USENIX FAST '04 Conference on File and Storage Technologies* (San Francisco, CA, March 2004), Hewlett-Packard Laboratories, USENIX Association, pp. 45–58.
 - [4] ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Information and control in gray-box systems. In *Proceedings of the 18th ACM Symp. on Operating Systems Principles* (Banff, Canada, October 2001), pp. 43–56.
 - [5] BUCY, J. S., GANGER, G. R., AND CONTRIBUTORS. The DiskSim simulation environment version 3.0 reference manual. Tech. Rep. CMU-CS-03-102, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 2003.
 - [6] DOUGLIS, F., KRISHNAN, P., AND BERSHAD, B. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing* (Ann Arbor, MI, April 1995), pp. 121–137.
 - [7] DOUGLIS, F., KRISHNAN, P., AND MARSH, B. Thwarting the power-hungry disk. In *Proceedings of 1994 Winter USENIX Conference* (San Francisco, CA, January 1994), pp. 292–306.
 - [8] HEATH, T., PINHEIRO, E., AND BIANCHINI, R. Application-supported device management for energy and performance. In *Proceedings of the 2002 Workshop on Power-Aware Computer Systems* (February 2002), pp. 114–123.
 - [9] HELMBOLD, D. P., LONG, D. D. E., AND SHERROD, B. A dynamic disk spin-down technique for mobile computing. In *Mobile Computing and Networking* (1996), pp. 130–142.
 - [10] HITACHI GLOBAL STORAGE TECHNOLOGIES. *Hitachi Micro-drive Hard Disk Drive Specifications*, January 2003.
 - [11] IEEE LOCAL AND METROPOLITAN AREA NETWORK STANDARDS COMMITTEE. *Wireless LAN medium access control (MAC) and physical layer (PHY) specifications*. New York, New York, 1997. IEEE Std 802.11-1997.
 - [12] INFORMATION SCIENCES INSTITUTE. NS-2 network simulator, 2003. <http://www.isi.edu/nsnam/ns/>.
 - [13] KIM, M., NOBLE, B., CHEN, X., AND TILBURY, D. Data propagation in a distributed file system, July 2004.
 - [14] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems* 10, 1 (February 1992).
 - [15] KRASHINSKY, R., AND BALAKRISHNAN, H. Minimizing energy for wireless web access with bounded slowdown. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking (MOBICOM '02)* (Atlanta, GA, July 2002).
 - [16] MUMMERT, L., EBLING, M., AND SATYANARAYANAN, M. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symp. on Op. Syst. Principles* (Copper Mountain, CO, Dec. 1995).
 - [17] MUMMERT, L., AND SATYANARAYANAN, M. Long term distributed file reference tracing: Implementation and experience. *Software Practice and Experience* 26, 6 (1996), 705–736.
 - [18] NETWORK WORKING GROUP. *NFS: Network File System protocol specification*, March 1989. RFC 1094.
 - [19] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 363–378.
 - [20] OUSTERHOUT, J. K., COSTA, H. D., HARRISON, D., KUNZE, J. A., KUFER, M., AND THOMPSON, J. G. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symp. on Op. Syst. Principles* (Orcas Island, WA, Dec. 1985), pp. 15–24.
 - [21] PAPATHANASIOU, A. E., AND SCOTT, M. L. Increasing disk burstiness through energy efficiency. Tech. Rep. 792, Computer Science Department, University of Rochester, November 2002.
 - [22] PAPATHANASIOU, A. E., AND SCOTT, M. L. Energy efficiency through burstiness. In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems and Applications* (Monterey, CA, October 2003), pp. 444–53.
 - [23] ROSENBLUM, M., BUGNION, E., HERROD, S. A., WITCHEL, E., AND GUPTA, A. The impact of architectural trends on operating system performance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)* (Copper Mountain, CO, December 1995), pp. 285–298.
 - [24] SIMUNIC, T., BENINI, L., GLYNN, P., AND MICHELI, G. D. Dynamic power management for portable systems. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MOBICOM '00)* (Boston, MA, August 2000), pp. 11–19.
 - [25] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technology* (San Francisco, CA, March/April 2003), pp. 72–88.
 - [26] TECHNOLOGIES, S. N. QualNet Simulator. <http://www.scalable-networks.com>.
 - [27] VOGELS, W. File system usage in Windows NT 4.0. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Kiawah Island, SC, December 1999), pp. 93–109.
 - [28] WEISSEL, A., BEUTEL, B., AND BELLOSA, F. Cooperative I/O: A novel I/O semantics for energy-aware applications. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 117–129.
 - [29] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th ACM Symp. on Op. Syst. Principles* (Copper Mountain, CO, Dec. 1995), pp. 96–108.
 - [30] ZEDLEWSKI, J., SOBTI, S., GARG, N., ZHENG, F., KRISHNAMURTHY, A., AND WANG, R. Modeling hard-disk power consumption. In *Proceedings of the 2nd USENIX Conference on File and Storage Technology* (San Francisco, CA, March/April 2003), pp. 217–230.
 - [31] ZENG, H., ELLIS, C. S., LEBECK, A. R., AND VAHDAT, A. Currency: A unifying abstraction for expressing energy management policies. In *Proceedings of the 2003 USENIX Annual Technical Conference* (San Antonio, TX, June 2003), pp. 43–56.

Itanium — A System Implementor's Tale

Charles Gray[†] Matthew Chapman[‡] Peter Chubb[‡] David Mosberger-Tang[§]

Gernot Heiser[‡]

[†] *The University of New South Wales, Sydney, Australia*

[‡] *National ICT Australia, Sydney, Australia*

[§] *HP Labs, Palo Alto, CA*

cgray@cse.unsw.edu.au

Abstract

Itanium is a fairly new and rather unusual architecture. Its defining feature is explicitly-parallel instruction-set computing (EPIC), which moves the onus for exploiting instruction-level parallelism (ILP) from the hardware to the code generator. Itanium theoretically supports high degrees of ILP, but in practice these are hard to achieve, as present compilers are often not up to the task. This is much more a problem for systems than for application code, as compiler writers' efforts tend to be focused on SPEC benchmarks, which are not representative of operating systems code. As a result, good OS performance on Itanium is a serious challenge, but the potential rewards are high.

EPIC is not the only interesting and novel feature of Itanium. Others include an unusual MMU, a huge register set, and tricky virtualisation issues. We present a number of the challenges posed by the architecture, and show how they can be overcome by clever design and implementation.

1 Introduction

Itanium [7] (also known as IA64) was introduced in 2000. It had been jointly developed by Intel and HP as Intel's architecture for the next decades. At present, Itanium processors are used in high-end workstations and servers.

Itanium's strong floating-point performance is widely recognised, which makes it an increasingly popular platform for high-performance computing. Its small-scale integer performance is so far less impressive. This is partially a result of integer performance being very dependent on the ability of the hardware to exploit any instruction-level parallelism (ILP) available in the code.

Most high-end architectures detect ILP in hardware, and re-order the instruction stream in order to maximise it. Itanium, by contrast, does no reordering, but instead relies on the code generator to identify ILP and represent it in the instruction stream. This is called *explicitly-parallel instruction-set computing* (EPIC), and is based on the established (but to date not overly successful)

very-long instruction word (VLIW) approach. EPIC is based on the realisation that the ILP that can be usefully exploited by reordering is limited, and aims at raising this limit.

The performance of an EPIC machine is highly dependent on the quality of the compiler's optimiser. Given the novelty of the architecture, it is not surprising that contemporary compilers are not quite up to the challenge [22]. Furthermore, most work on compilers is focusing on application code (in fact, mostly on SPEC benchmarks), so compilers tend to perform even worse on systems code. Finally, of the various compilers around, by far the weakest, GCC, is presently the default for compiling the Linux kernel. This poses a number of challenges for system implementors who strive to obtain good OS performance on Itanium.

Another challenge for the systems implementor is presented by Itanium's huge register file. This helps to keep the pipelines full when running CPU-bound applications, but if all those registers must be saved and restored on a context switch, the costs will be significant, Itanium's high memory bandwidth notwithstanding. The architecture provides a *register stack engine* (RSE) which automatically fills/spills registers to memory. This further complicates context switches, but has the potential for reducing register filling/spilling overhead [21]. The large register set, and the mechanisms for dealing with it, imply trade-offs that lead to different implementation strategies for a number of OS services, such as signal handling.

Exceptions are expensive on processors with high ILP and deep pipelines, as they imply a break in the execution flow that requires flushing the pipeline and wasting many issue slots. For most exceptions this is unavoidable but irrelevant if the exceptions are relatively infrequent (like interrupts) or a result of program faults. System calls, however, which are treated as exceptions on most architectures, are not faults nor necessarily infrequent, and must be fast. Itanium deals with this issue by providing a mechanism for increasing the privilege level without an exception and the corresponding

pipeline flush, but it is subject to limitations which make it tricky to utilise.

Itanium's memory-management unit (MMU) also has some unusual properties which impact on OS design. Not only does it support a wide range of page sizes (which is nothing unusual), it also supports the choice of two different hardware page-table formats, a virtual linear array (called *short VHPT* format) and a hash table (called the *long VHPT* format). As the names imply, they have different size page table entries, and different performance and feature tradeoffs, including the support for superpages and the so-called *protection keys*. The hardware page-table walker can even be disabled, effectively producing a software-loaded TLB.

Protection keys loosen the usual nexus between protection and translation: access rights on pages are not only determined by access bits on page-table entries, but also by an orthogonal mechanism which allows grouping sets of pages for access-control purposes. This mechanism also supports sharing of a single entry in the translation lookaside buffer (TLB) between processes sharing access to the page, even if their access rights differ.

The original architecture is disappointing in a rather surprising respect: it is not fully virtualisable. Virtual-machine monitors (VMMs) have gained significant popularity in recent years, and Itanium is almost, but not quite, virtualisable. This creates a significant challenge for anyone who wants to develop an Itanium VMM. Fortunately, Intel recognised the deficiency and is addressing it with an architecture-extension called Vanderpool Technology [10], which is to be implemented in future CPUs.

This paper presents a discussion of the features of the Itanium architecture which present new and interesting challenges and design tradeoffs to the system implementor. We will discuss the nature of those challenges, and how they can be dealt with in practice. First, however, we present an overview of the Itanium architecture in the next section. In Section 3 we discuss the most interesting features of the Itanium's memory-management unit and the design tradeoffs it implies. In Section 4 we discuss issues with virtualisation of Itanium, while Section 5 presents a number of case studies of performance tradeoffs and micro-optimisation. Section 6 concludes the paper.

2 Itanium Architecture Overview

2.1 Explicitly-parallel instruction-set computing

As stated in the Introduction, Itanium's EPIC approach is based on VLIW principles, with several instructions contained in each instruction word. Scheduling of in-

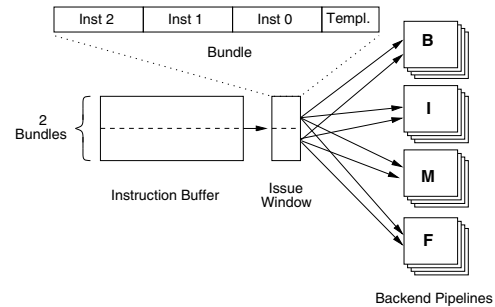


Figure 1: Instruction Issue

structions, and specification of ILP, becomes the duty of the compiler (or assembly coder). This means that details of the processor pipelines and instruction latencies must be exposed in the architecture, so the compiler can emit correct code without the processor needing to scoreboard instruction dependencies.

The Itanium approach to EPIC aims at achieving this without overly limiting the design space of future processors, i.e., by describing ILP in a way that does not depend on the actual number of pipelines and functional units. The compiler is encouraged to maximise ILP in the code, in order to optimise performance for processors regardless of pipeline structure. The result is a greatly simplified instruction issue, with only a few pipeline stages dedicated to the processor front-end (two front-end and six back-end stages, ignoring floating point, for Itanium 2). The shorter pipeline helps to reduce exception and mis-prediction penalties.

Itanium presents a RISC-like load/store instruction set. Instructions are grouped into 128-bit *bundles*, which generally hold three instructions each. Several bundles form an *instruction group* delimited by *stops*. Present Itanium processors use a two-bundle issue window (resulting in an issue of six instructions per cycle). By definition, all instructions in a group are independent and can execute concurrently (subject to resource availability).

Figure 1 shows the first few stages of the Itanium pipeline. Bundles are placed into the instruction buffer speculatively and on demand. Each clock cycle, all instructions in the issue window are dispersed into backend pipelines (branch, memory, integer and floating-point) as directed by the *template*, unless a required pipeline is stalled or a stop is encountered in the instruction stream.

Each bundle has a 5-bit template field which specifies which instructions are to be dispersed into which pipeline types, allowing the instruction dispersal to be implemented by simple static logic. If there are not enough backend units of a particular type to disperse an instruction, *split issue* occurs; the preceding instructions

are issued but that instruction and subsequent instructions must wait until the next cycle — Itanium issues strictly in order. This allows a compiler to optimise for a specific processor based on the knowledge of the number of pipelines, latencies etc., without leading to incorrect execution on earlier or later processors.

One aspect of EPIC is to make even *data* and *control speculation* explicit. Itanium supports this through *speculative load* instructions, which the compiler can move forward in the instruction stream without knowing whether this is safe to do (the load could be through an invalid pointer or the memory location overwritten through an alias). Any exception resulting from a speculative load is deferred until the result is consumed. In order to support speculation, general registers are extended by an extra bit, the *NaT* (“not a thing”) bit, which is used to trap mis-speculated loads.

2.2 Register stack engine

Itanium supports EPIC by a huge file of architected registers, rather than relying on register renaming in the pipeline. There are 128 user-mode *general registers* (GRs), the first 32 of which are *global*; 16 of these are banked (i.e., there is a separate copy for privileged mode). The remaining 96 registers are explicitly renamed by using register windows, similar to the SPARC [23].

Unlike the SPARC’s, Itanium’s register windows are of variable size. A function uses an `alloc` instruction to allocate *local* and *output* registers. On a function call via the `br.call` instruction, the window is rotated up past the local registers leaving only the caller’s output registers exposed, which become the callee’s *input* registers. The callee can then use `alloc` to widen the window for new local and output registers. On executing the `br.ret` instruction, the caller’s register window is restored.

The second, and most important, difference to the SPARC is the Itanium’s *register stack engine* (RSE), which transparently spills or fills registers from memory when the register window overflows or underflows the available registers. This not only has the advantage of freeing the program from dealing with register-window exceptions. More importantly, it allows the processor designers to transparently add an arbitrary number of windowed registers, beyond the architected 96, in order to reduce memory traffic from register fills/spills. It also supports lazy spilling and pre-filling by the hardware.

Internally, the stack registers are partitioned into four categories — current, dirty, clean and invalid. *Current* registers are those in the active procedure context. *Dirty* registers are those in a parent context which have not yet been written to the backing store, while *clean* registers are parent registers with valid contents that have been

written back (and can be discarded if necessary). *Invalid* registers contain undefined data and are ready to be allocated or filled.

The RSE operation is supported by a number of special instructions. The `flushrs` instruction is used to force the dirty section of registers to the backing store, as required on a context switch. Similarly, the `loadrs` instruction is used to reload registers on a context switch. The `cover` instruction is used to allocate an empty register frame above the previously allocated frame, ensuring any previous frames are in the dirty or clean partitions.

There is another form of register renaming: *register rotation*, which rotates registers within the current register window. This is used for so-called *software pipelining* and supports optimisations of tight loops. As this is mostly relevant at application level it is not discussed further in this paper.

2.3 Fast system calls

Traditionally, a system call is implemented by some form of invalid instruction exception that raises the privilege level, saves some processor state and diverts to some handler code. This is essentially the same mechanism as an interrupt, except that it is synchronous (triggered by a specific instruction) and therefore often called a *software interrupt*.

Such an exception is inherently expensive, as the pipeline must be flushed, and speculation cannot be used to mitigate that cost. Itanium provides a mechanism for raising the privilege level without an exception, based on *call gates*. The MMU supports a special permission bit which allows designating a page as a *gate page*. If an *epc* instruction in such a page is executed, the privilege level is raised without any other side effects. Code in the call page (or any code jumped to once in privileged mode) can access kernel data structures and thus implement system calls. (Other architectures, such as IA-32, also provide gates. The Itanium version is more tricky to use, see Section 5.2).

2.4 Practical programming issues

The explicit management of ILP makes Itanium performance critically dependent on optimal scheduling of instructions in the executable code, and thus puts a stronger emphasis on compiler optimisation (or hand-optimised assembler) than other architectures. In this section we discuss some of these issues.

2.4.1 Bundling and latencies

The processor may issue less than a full (six instruction) issue window in a number of cases (*split issue*). This can happen if the instructions cannot be issued concurrently due to dependencies, in which case the compiler

inserts *stops* which instruct the processor to split issue. Additionally, split issue will occur if the number of instructions for a particular functional unit exceeds the (processor-dependent) number of corresponding back-end units available. Split issue may also occur in a number of processor-specific cases. For example, the Itanium 2 processor splits issue directly after serialisation instructions (*srlz* and *sync*).

Optimum scheduling also depends on accurate knowledge of instruction latency, defined as the number of cycles of separation needed between a producing instruction and a consuming instruction. Scheduling a consuming instruction within less than the producing instruction's latency does not lead to incorrect results, but stalls execution not only of this instruction, but also of all in the current and subsequent instruction-groups.

ALU instructions as well as load instructions that hit in the L1 cache have single-cycle latencies. Thus the great majority of userspace code can be scheduled without much consideration of latencies — one simply needs to ensure that consumers are in instruction groups subsequent to producers.

However, the situation is different for system instructions, particularly those accessing *control registers* and *application registers*. On the Itanium 2 processor, many of these have latencies of 2–5 cycles, a few (processor-state register, RSE registers and kernel registers) have latencies of 12 cycles, some (timestamp counter, interrupt control and performance monitoring registers) have 36 cycle latencies. This makes scheduling of systems code difficult, and the performance cost of getting it wrong very high.

2.4.2 Other pipeline stalls

Normally latencies can be dealt with by overlapping execution of several bundles (Itanium supports out-of-order completion). However, some instructions cannot be overlapped, producing unconditional stalls. This naturally includes the various serialisation instructions (*srlz*, *sync*) but also instructions that force RSE activity (*flushrs*, *loadrs*). Exceptions and the *rfi* (return from exception) instruction also produce unavoidable stalls, but these can be avoided for system calls by using *epc*.

There also exist other constraints due to various resource limitations. For example, while stores do not normally stall, they consume limited resources (store buffers and L2 request queue entries) and can therefore stall if too many of them are in progress. Similarly, the high-latency accesses to privileged registers are normally queued to avoid stalls and allow overlapped execution. However, this queue is of limited size (8 entries on Itanium 2); only one result can be returned per cycle, and the results compete with loads for writeback

resources. Moreover, accesses to the particularly slow registers (timestamp counter, interrupt control and performance monitoring registers) can only be issued every 6 cycles.

A case study of minimising stalls resulting from latencies in system code is given in Section 5.3.

3 Memory-Management Unit

3.1 Address translation and protection

As mentioned earlier, the memory-management unit (MMU) of the Itanium has a number of unusual features. The mechanics of address translation and access-right lookup are schematically shown in Figure 2. The top three bits of the 64-bit virtual address form the *virtual region number*, which is used to index into a set of eight *region registers* (RRs) which contain *region IDs*.

The remaining 61 bits form the *virtual page number* (VPN) and the *page offset*. Itanium 2 supports a wide range of page sizes, from 4kB to 4GB. The VPN is used together with the region ID to perform a fully-associative lookup of the *translation lookaside buffer* (TLB). The region ID serves as a generalisation of the *address-space ID* (ASID) tags found on many RISC processors.

Like an ASID, the region ID supports the co-existence of mappings from different contexts without causing aliasing problems, but in addition allows for simple sharing of pages on a per-region basis: if two processes have the same region ID in one of their RRs, they share all mappings in that region. This provides a convenient way for sharing text segments, if one region is reserved for program code and a separate region ID is associated with each executable. Note that if region IDs are used for sharing, the processes not only share pages, but actually share the TLB entries mapping those pages. This helps to reduce TLB pressure.

A more unusual feature of the Itanium TLB is the *protection key* tag on each entry (which is a generalisation of the *protection-domain identifiers* of the PA-RISC [24]). If protection keys are enabled, then the key field of the matching TLB entry is used for an associative lookup of another data structure, a set of *protection key registers* (PKRs). The PKR contains a set of access rights which are combined with those found in the TLB to determine the legality of the attempted access. This can be used to implement write-only mappings (write-only mode is not supported by the rights field in the TLB).

Protection keys can be used to share individual (or sets of) pages with potentially different access rights. For example, if two processes share a page, one process with read-write access, the other read-only, then the page can be marked writable in the TLB, and given a protection key. In the one process's context, the rights field in the

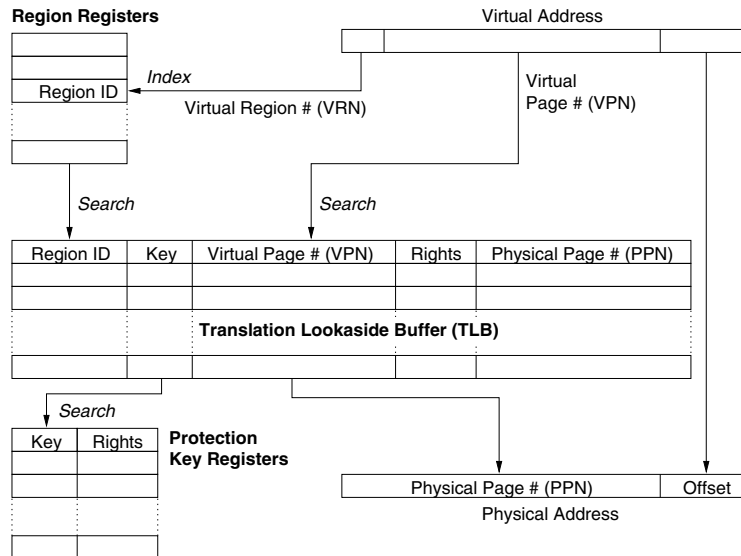


Figure 2: Itanium address translation and memory protection.

corresponding PKR would be set to read-write, while for the other process it would be set to read-only. The processes again share not only the page but also the actual TLB entries. The OS can even use the rights field in the TLB to downgrade access rights for everybody, e.g. for implementing copy-on-write, or for temporarily disabling all access to the page.

3.2 Page tables

The Itanium has hardware support for filling the TLB by walking a page table called the *virtual hashed page table* (VHPT). There are actually two hardware-supported page-table formats, called the *short-format* and *long-format* VHPT respectively. The hardware walker can also be completely turned off, requiring all TLB reloads to be done in software (from an arbitrary page table structure).

Turning off the hardware walker is a bad idea. We measured the average TLB refill cost in Linux to be around 45 cycles on an Itanium 2 with the hardware walker enabled, compared to around 160 cycles with the hardware walker disabled. A better way of supporting arbitrary page table formats is to use the VHPT as a hardware-walked software TLB [2] and reload from the page table proper on a miss.

Figure 3 shows the format and access of the two types of page table. The short-format VHPT is, name notwithstanding, a *linear virtual array page table* [5, 12] that is indexed by the page number and maps a single region, hence up to eight are required per process, and the size of each is determined by the page size. Each page table entry (PTE) is 8 bytes (one word) long. It contains a physical page number, access rights, caching at-

tributes and software-maintained *present*, *accessed* and *dirty* bits, plus some more bits of information not relevant here. A region ID need not be specified in the short VHPT, as it is implicit in the access (each region uses a separate VHPT).

The page size is also not specified in the PTE, instead it is taken from the *preferred page size* field contained in the region register. This implies that when using the short VHPT, the hardware walker can be used for only one page size per region. Non-default page-sizes within a region would have to be handled by (slower) software fills.

The PTE also contains no protection key, instead the architecture specifies that the protection key is taken from the corresponding region register (and is therefore the same as the region ID, except that the two might be of different length). This makes it impossible to specify different protection keys in a region if the short-format VHPT is used. Hence, sharing TLB entries of selected (shared) pages within a region is not possible with this page table format.

The long VHPT is a proper hashed page table, indexed by a hash of the page number. Its size can be an arbitrary power of two (within limits), and a single table can be used for all regions. Its entries are 32 bytes (4 words) long and contain all the information of the short VHPT entries, plus a page-size specification, a protection key, a tag and a chain field. Hence, the long VHPT supports a per-page specification of page size and protection key. The tag field is used to check for a match on a hashed access and must be generated by specific instructions. The chain field is ignored by the hardware and can be used by the operating system to implement overflow chains.

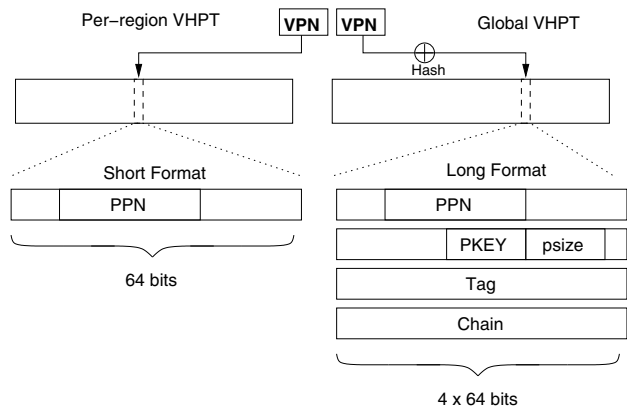


Figure 3: Short and long VHPT formats.

3.3 VHPT tradeoffs

The advantage of the short VHPT is that its entries are compact and highly localised. Since the Itanium's L1 cache line size is 64 bytes, a cache line can hold 8 short entries, and as they form a linear array, the mappings for neighbouring pages have a high probability of lying in the same cache line. Hence, locality in the page working set translates into very high locality in the PTEs, and the number of data cache lines required for PTEs is small.

In contrast, a long VHPT entry is four times as big, and only two fit in a cache line. Furthermore, hashing destroys locality, and hence the probability of two PTEs sharing a cache line is small, unless the page table is small and the page working set large (a situation which will result in collisions and expensive software walks). Hence, the long VHPT format tends to be less cache-friendly than the short format.

The long-format VHPT makes up for this by being more TLB friendly. For the short format, at least three TLB entries are generally required to map the page table working set of each process, one for code and data, one for shared libraries and one for the stack. Linux in fact, typically uses three regions for user code, and thus will require at least that many entries for mapping a single process's page tables. In contrast, a process's whole long-format VHPT can be mapped with a single large superpage mapping. Furthermore, a single long-format VHPT can be shared between all processes, reducing TLB entry consumption for page tables from ≥ 3 per process to one per CPU.

This tradeoff is likely to favour the short-format VHPT in cases where TLB pressure is low, i.e., where the total page working set is smaller than the TLB capacity. This is typically the case where processes have mostly small working sets and context switching rates are low to moderate. Many systems are likely to operate in that regime, which is the reason why present Linux only supports the short VHPT format.

The most important aspect of the two page table formats is that the short format does not support many of the Itanium's MMU features, in particular hardware-loaded mixed page sizes (superpages) within a region. Superpages have been shown to lead to significant performance improvements [17] and given the overhead of handling TLB-misses in software, it is desirable to take advantage of the hardware walker. As Linux presently uses the short-format VHPT, doing so would require a switch of the VHPT format first. This raises the question whether the potential performance gain might be offset by a performance loss resulting from the large page-table format.

3.4 Evaluation

We did a comparison of page-table formats by implementing the long-format VHPT in the Linux 2.6.6 kernel. We ran the *lmbench* [15] suite as well as Suite IX of the *aim* benchmark [1], and the OSDL DBT-2 benchmark [18]. Tests were run on a HP rx2600 server with dual 900MHz Itanium-2 CPUs. The processors have three levels of on-chip cache. The L1 is a split instruction and data cache, each 16kB, 4-way associative with a line size of 64 bytes and a one-cycle hit latency. The L2 is a unified 256kB 8-way associative cache with 128B lines and a 5 cycle hit latency. The L3 is 1.5MB large, 6-way associative, with a 128B line size and 12 cycles hit latency. The memory latency with the HP zx1 chipset is around 100 cycles.

The processors have separate fully-associative data and instruction TLBs, each structured as two-level caches with 32 L1 and 128 L2 entries. Using 16kB pages, the per-CPU long-format VHPT was sized at 16MB in our experiments, being four times the size needed to map the entire 2G physical memory.

The results for the *lmbench* process and file-operation benchmarks are uninteresting. They show that the choice of page table has little impact on performance. This is not very surprising, as for these benchmarks there is no significant space pressure on either the CPU caches or the TLB.

Somewhat more interesting are the results of the *lmbench* context-switching benchmarks, shown in Table 1. Here the long-format page table shows some noticeable performance advantage with a large number of processes but small working sets (and consequently high context-switching rates). This is most likely a result of the long-format VHPT reducing TLB pressure. The performance of the two systems becomes equal again when the working sets increase, probably a result of the better cache-friendliness of the short-format page table, and the reduced relative importance of TLB miss handling costs.

The other *lmbench* runs as well as the *aim* bench-

Context switching with 0K							
	2proc	4proc	8proc	16proc	32proc	64proc	96proc
U	0.98	1.00	0.95	0.88	0.98	1.44	1.34
M	0.94	0.96	0.95	0.96	1.23	1.30	1.27

Context switching with 4K							
	2proc	4proc	8proc	16proc	32proc	64proc	96proc
U	0.97	0.99	0.97	0.95	1.17	1.20	1.09
M	0.95	0.61	0.78	0.87	1.11	1.13	1.09

Context switching with 8K							
	2proc	4proc	8proc	16proc	32proc	64proc	96proc
U	0.99	0.98	0.96	0.97	1.31	1.17	1.08
M	0.95	0.91	0.96	1.00	1.29	1.15	1.06

Context switching with 16K							
	2proc	4proc	8proc	16proc	32proc	64proc	96proc
U	0.99	0.98	0.96	0.97	1.31	1.17	1.08
M	0.95	0.91	0.96	1.00	1.29	1.15	1.06

Context switching with 32K							
	2proc	4proc	8proc	16proc	32proc	64proc	96proc
U	0.98	0.99	1.04	1.30	1.04	1.03	1.00
M	0.94	0.96	1.00	1.01	0.87	1.00	1.00

Context switching with 64K							
	2proc	4proc	8proc	16proc	32proc	64proc	96proc
U	1.00	0.98	0.94	0.94	1.00	1.00	1.00
M	0.97	0.98	1.06	1.22	0.94	0.99	0.98

Table 1: Lmbench context-switching results. Numbers indicate performance with a long-format VHPT relative to the short-format VHPT: a figure > 1.0 indicates better, < 1.0 worse performance than the short-format page table. Lines marked “U” are for a uniprocessor kernel, while “M” is the same for a multiprocessor kernel (on a two-CPU system).

mark results were similarly unsurprising and are omitted for space reasons. Complete results can be found in a technical report [4].

The SPEC CPU2000 integer benchmarks, AIM7 and lmbench show no cases where the long-format VHPT resulted in significantly worse performance than the short-format VHPT, provided the long-format VHPT is sized correctly (with the number of entries equal to four times the number of page frames).

We also ran OSDL’s DBT-2 benchmark, which emulates a warehouse inventory system. This benchmark stresses the virtual memory system — it has a large resident set size, and has over 30 000 TLB misses per second. The results show no significant performance difference at an 85% confidence level — for five samples, the long format VHPT gave 400(6) transactions per minute, and the short format page table gave 401(4) transactions per minute (standard deviation in the parentheses).

We also investigated TLB entry sharing, but found no significant benefits with standard benchmarks [4].

Based on these experiments, we conclude that long-format VHPT can provide performance as good or better than short-format VHPT. Given that long-format VHPT also enables hardware-filled superpages and TLB-entry-sharing across address-spaces, we believe it may very well make sense to switch Linux to the long-format VHPT in the future.

4 Virtualisation

Virtualisability of a processor architecture [20] generally depends on a clean separation between *user* and *system* state. Any instructions that inspect or modify the system state (*sensitive instructions*) must be privileged, so that the VMM can intervene and emulate their behaviour with respect to the simulated machine. Some exceptions to this may be permissible where the virtual machine monitor can ensure that the real state is synchronised with the simulated state.

In one sense Itanium is simpler to virtualise than IA-32, since most of the instructions that inspect or modify system state are privileged by design. It seems likely that the original Itanium designers believed in this clear separation of user and system state which is necessary for virtualisation. Sadly, a small number of non-virtualisable features have crept into the architecture, as we discovered in our work on the vNUMA distributed virtual machine [3]. Some of these issues were also encountered by the authors of vBlades [14], a recent virtual machine for the Itanium architecture.

The `cover` instruction creates a new empty register stack frame, and thus is not privileged. However, when executed with interruption collection off (interruption collection controls whether execution state is saved to the interruption registers on an exception), it has the side-effect of saving information about the previous stack frame into the privileged *interruption function state* (IFS) register. Naturally, it would not be wise for a virtual machine monitor to actually turn off interruption collection at the behest of the guest operating system, and when the simulated interruption collection bit is off, there is no way for it to intercept the `cover` instruction and perform the side-effect on the simulated copy of IFS. Hence, `cover` must be replaced with an instruction that faults to the VMM, either statically or at run time.

The `thash` instruction, given an address, calculates the location of the corresponding hashtable entry in the VHPT. The `ttag` instruction calculates the corresponding tag value. These instructions are, for some reason, unprivileged. However, they reveal processor memory management state, namely the pagetable base, size and format. When the guest OS uses these instructions, it obtains information about the real pagetables instead of its own pagetables. Therefore, as with `cover`, these instructions must be replaced with faulting versions.

Virtual memory semantics also need to be taken into account, since for a virtual machine to have reasonable performance, the majority of virtual memory accesses need to be handled by the hardware and should not trap to the VMM. For the Itanium architecture, most features can be mapped directly. However, a VMM will need to reserve some virtual address space (at least for exception handlers). One simple way to do this is to report a smaller virtual address space than implemented on the real processor, thereby ensuring that the guest operating system will not use certain portions. On the other hand, the architecture defines a fixed number of privilege levels (0 to 3). Since the most privileged level must be reserved for the VMM, this means that the four privilege levels in the guest must be mapped onto three real privilege levels (a common technique known as *ring compression*). This means there may be some loss of protection, though most operating systems do not use all four privilege levels.

The Itanium architecture provides separate control over instruction translation, data translation and register-stack translation. For example, it is possible to have register-stack translation on (virtual) and data translation off (physical). There is no way to efficiently replicate this in virtual mode, since register-stack references and data references access the same virtual address space.

Finally, if a fault is taken while the register-stack engine is filling the current frame, the RSE is halted and the exception handler is executed with an incomplete frame. As soon as the exception handler returns, the RSE resumes trying to load the frame. This poses difficulties if the exception handler needs to return to the guest kernel (at user-level) to handle the fault.

Future Itanium processors will have enhanced virtualisation support known as Vanderpool Technology. This provides a new processor operating mode in which sensitive instructions are properly isolated. Additionally, this mode is designed so as to allow the guest operating system to run at its normal privilege level (0) without compromising protection, negating the need for ring compression. Vanderpool Technology also provides facilities for some of the virtualisation to be handled in hardware or firmware (*virtualisation acceleration*). In concert these features should provide for simpler and more efficient virtualisation. Nevertheless, there remain some architectural features which are difficult to virtualise efficiently and require special treatment, in particular the translation modes and the RSE issue described above.

5 Case studies

In this section we present three implementation studies which we believe are representative of the approaches that need to be taken to develop well-performing sys-

tems software on Itanium. The first example, implementation of signals in Linux, illustrates that Itanium features (in this case, the large register file) lead to different tradeoffs from these on other architectures. The second example investigates the use of the fast system-call mechanism in Linux. The third, micro-optimisation of a fast system-call path, illustrates the challenges of EPIC (and the cost of insufficient documentation).

5.1 Efficient signal delivery

In this section we explore a technique to accelerate signal delivery in Linux. This is an exercise in intelligent state-management, necessitated by the large register file of the Itanium processor, and relies heavily on exploiting the software conventions established for the Itanium architecture [8]. The techniques described here not only improved signal-delivery performance on Itanium Linux, but also simplified the kernel.

In this section we use standard Itanium terminology. We use *scratch register* to refer to a caller-saved register, i.e., a register whose contents is *not* preserved across a function-call. Analogously, we use *preserved register* to refer to a callee-saved register, i.e., a register whose contents *is* preserved across a function-call.

5.1.1 Linux signal delivery

The canonical way for delivering a signal in Linux consists of the following steps:

- On any entry into the kernel (e.g., due to system call, device interrupt, or page-fault), Linux saves the scratch registers at the top of the kernel-stack in a structure called *pt_regs*.
- Right before returning to user level, the kernel checks whether the current process has a signal pending. If so, the kernel:
 1. saves the contents of the *preserved* registers on the kernel-stack in a structure called *switch_stack* (on some architectures, the *switch_stack* structure is an implicit part of *pt_regs* but for the discussion here, it's easier to treat it as separate);
 2. calls the routine to deliver the signal, which may ignore the signal, terminate the process, create a core dump, or arrange for a signal handler to be invoked.

The important point here is that the combination of the *pt_regs* and *switch_stack* structures contain the full user-level state (machine context). The *pt_regs* structure obviously contains user-level state, since it is created right on entry to the kernel. For the *switch_stack* structure, this is also true but less obvious: it is true because at the time the *switch_stack* structure is created, the kernel stack is empty apart from

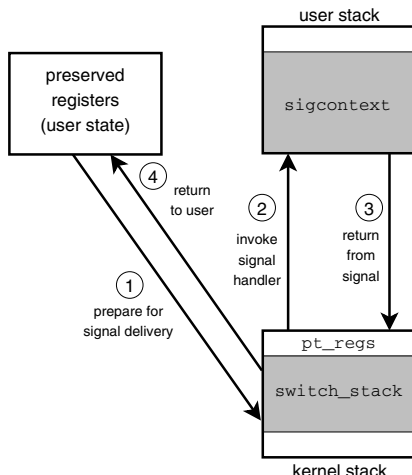


Figure 4: Steps taken during signal delivery

the `pt_regs` structure. Since there are no intermediate call frames, the preserved registers must by definition contain the original user-level state.

Signal-delivery requires access to the full user-level state for two reasons:

1. if the signal results in a core dump, the user-level state needs to be written to the core file;
2. if the signal results in the invocation of a signal handler, the user-level state needs to be stored in the `sigcontext` structure.

5.1.2 Performance Considerations

The problem with the canonical way of delivering a signal is that it entails a fair amount of redundant moving of state between registers and memory. For example, as illustrated in Figure 4, the preserved registers:

1. get saved on the kernel stack in preparation for signal delivery
2. get copied to the user-level stack in preparation for invoking a signal handler
3. get copied back to the kernel stack on return from a signal-handler
4. need to be restored from the kernel-stack upon returning execution to user level.

On architectures with small numbers of architected registers, redundant copying of registers is not a big issue, particularly since their contents is likely to be hot in the cache anyway. However, with Itanium's large register file, the cost of copying registers can be high.

When faced with this challenge, we decided that rather than trying to micro-optimize the moving of the state, a better approach would be to *avoid* the redundant moves in the first place. This was helped by the following observations:

- For a core dump, the preserved registers can be reconstructed after the fact with the help of a kernel-stack unwinder. Specifically, when the kernel needs to create a core dump, it can take a snapshot of the current registers and then walk the kernel stack. In each stack frame, it can update the snapshot with the contents of the registers saved in that stack frame. When reaching the top of the kernel-stack, the snapshot contains the desired user-level state.
- There is no *inherent* reason for saving the preserved registers in the `sigcontext` structure. While it is *customary* to do so, there is nothing in the Single-UNIX Specification [19] or the POSIX standard that would require this. The reason it is not necessary to include the preserved registers in the `sigcontext` structure is that the signal handler (and its callees) automatically save preserved registers before using them and restore them before returning. Thus, there is no need to create a copy of these registers in the `sigcontext` structure. Instead, we can just leave them alone.

In combination, these two observations make it possible to completely eliminate the `switch_stack` structure from the signal subsystem.

We made this change for Itanium Linux in December 2000. At that time, there was some concern about the existence of applications which rely on having the full machine-state available in `sigcontext` and for this reason, we left the door open for a user-level compatibility-layer which would make it appear as if the kernel had saved the full state [16]. Fortunately, in the four years since making the change, we have not heard of a need to activate the compatibility layer.

To quantify the performance effect of saving only the minimal state, we forward-ported the original signal-handling code to a recent kernel (v2.6.9-rc3) and found it to be 23–34% slower. This relative slowdown varied with kernel-configuration (uni- vs. multi-processor) and chip generation (Itanium vs. Itanium 2). The absolute slowdown was about 1,400 cycles for Itanium and 700 cycles for Itanium 2. We should point out that, had it not been for backwards-compatibility, `sigcontext` could have been shrunk considerably and fewer cache-lines would have to be touched during signal delivery. In other words, in a design free from compatibility concerns, the savings could be even bigger.

Table 2 shows that saving the minimal state yields signal-delivery performance that is comparable to other architectures: even a 1GHz Itanium 2 can deliver signals about as fast as a 2.66GHz Pentium 4.

Apart from substantially speeding up signal delivery, this technique (which is not Itanium-specific) simplified the kernel considerably: it eliminated the need to maintain the `switch_stack` in the signal subsystem and

Chip		SMP		UP	
		cycles	μ s	cycles	μ s
Itanium 2	1.00 GHz	3,087	3.1	2,533	2.5
Pentium 4	2.66 GHz	8,320	3.2	6,500	2.4

Table 2: Signalling times with Linux kernel v2.6.9-rc3. (SMP = multiprocessor kernel, UP = uniprocessor kernel)

System Call	Dynamic		Static	
	break	epc	break	epc
getpid()	294	18	287	12
getppid()	299	77	290	54
gettimeofday()	442	174	432	153

Table 3: Comparison of system call costs (in cycles) using the standard (break) and fast (epc) mechanism, both with dynamically and statically linked binaries

removed all implicit dependencies on the existence of this structure.

5.2 Fast system-call implementation

5.2.1 Fast system calls in Linux

As discussed in Section 2.3, Itanium provides gate pages and the `epc` instruction for getting into kernel mode without a costly exception. Here we discuss the practicalities of using this mechanism in Linux.

After executing the `epc` instruction, the program is executing in privileged mode, but still uses the user's stack and register-backing store. These cannot be trusted by the kernel, and therefore such a system call is very limited, until it loads a sane stack and RSE backing-store pointer. This is presently not supported in Linux, and thus the fast system-call mechanism is restricted by the following conditions:

- the code cannot perform a normal function call (which would create a new register stack frame and could lead to a spill to the RSE backing store);
- the code must not cause an exception, because normal exception handling spills registers. This means that all user arguments must be carefully checked, including checking for a possible NaT consumption exception (which could normally be handled transparently).

As a result, fast system calls are presently restricted to handcrafted assembly language code, and functionality that is essentially limited to passing data between the kernel and the user. System calls fitting those requirements are inherently short, and thus normally dominated by the exception overhead, so good candidates for implementing in an exception-less way.

So far we implemented the trivial system calls `getpid()` and `getppid()`, and the

somewhat less trivial `gettimeofday()`, and `rt_sigprocmask()`. The benefit is significant, as shown in Table 3: we see close to a factor of three improvement for the most complicated system call. The performance of `rt_sigprocmask()` is not shown. Currently glibc does not implement `rt_sigprocmask()`, so it is not possible to make a meaningful comparison.

5.3 Fast message-passing implementation

Linux, owing to its size and complexity, is not the best vehicle for experimenting with fast system calls. The L4 microkernel [11] is a much simpler platform for such work, and also one where system-call performance is much more critical. Message-passing inter-process communication (IPC) is the operation used to invoke any service in an L4-based system, and the IPC operation is therefore highly critical to the performance of such systems. While there is a generic (architecture-independent) implementation of this primitive, for the common (and fastest) case it is usually replaced in each port by a carefully-optimised architecture-specific version. This so-called *IPC fast path* is usually written in assembler and tends to be of the order of 100 instructions. Here we describe our experience with micro-optimising L4's IPC operation.

5.3.1 Logical control flow

The logical operation of the IPC fast path is as follows, assuming that a sender invokes the `ipc()` system call and the receiver is already blocked waiting to receive:

1. enter kernel mode (using `epc`);
2. inspect the *thread control blocks* (TCBs) of source and destination threads;
3. check that fast path conditions hold, otherwise call the generic "slow path" (written in C++);
4. copy message (if the whole message does not fit in registers);
5. switch the register stack and several other registers to the receiver's state (most registers are either used to transfer the message or clobbered during the operation);
6. switch the address space (by switching the page table pointer);
7. update some state in the TCBs and the pointer to the current thread;
8. return (in the receiver's context).

The original implementation of this operation (a combination of C++ code, compiled by GCC 3.2, and some assembly code to implement context switching) executed in 508 cycles with hot caches on an Itanium-2 machine. An initial assembler fast path to transfer up to 8 words, only loosely optimised, brought this down to 170

```

56    BACK_END_BUBBLE.ALL
30    BE_EXE_BUBBLE.ALL
16    BE_EXE_BUBBLE.GRALL
14    BE_EXE_BUBBLE.ARCR
15    BE_L1D_FPU_BUBBLE.ALL
10    BE_L1D_FPU_BUBBLE.L1D_DCURECIR
5     BE_L1D_FPU_BUBBLE.L1D_STBUFRECIR
11    BE_RSE_BUBBLE.ALL
4     BE_RSE_BUBBLE.AR_DEP
6     BE_RSE_BUBBLE.LOADRS
1     BE_RSE_BUBBLE.OVERFLOW

```

Figure 5: Breakdown of bubbles provided by the PMU.

cycles. While this is a factor of three faster, it is still on the high side; on RISC architectures the operation tends to take 70–150 cycles [13].¹

5.3.2 Manual optimisation

An inspection of the code showed that it consisted of only 83 instruction groups, hence 87 cycles were lost to bubbles. Rescheduling instructions to eliminate bubbles would potentially double performance!

An attempt at manual scheduling resulted not only in an elimination of bubbles, but also a reduction of the number of instruction groups (mostly achieved by rearranging the instructions to make better use of the available instruction templates). The result was 39 instruction groups executing in 95 cycles. This means that there were still 56 bubbles, accounting for just under 60% of execution time.

The reason could only be that some instructions had latencies that were much higher than expected. Unfortunately, Intel documentation contains very little information on instruction latencies, and did not help us further.

Using the *perfmon* utility [6] to access Itanium’s *performance monitoring unit* (PMU) we obtained the breakdown of the bubbles summarised in Figure 5. The data in the figure is to be read as follows: 56 bubbles were recorded by the counter `back_end_bubble.all`. This consists of 30 bubbles for `be_exe_bubble.all`, 15 bubbles for `be_l1d_fpu_bubble.all` and 11 bubbles for `be_rse_bubble.all`. Each of these is broken down further as per the figure.

Unfortunately, the Itanium 2 Processor Reference Manual [9] is not very helpful here, it typically gives a one-line summary for each PMU counter, which is insufficient to understand what is happening. What was clear, however, was the register stack engine was a significant cause of latency.

5.3.3 Fighting the documentation gap

Register-stack-engine stalls In order to obtain the information required to optimise the code further, we saw

no alternative to systematically measuring the latencies between any two instructions which involve the RSE. The results of those measurements are summarised in Table 4. Some of those figures are surprising, with some seemingly innocent instructions having latencies in excess of 10 cycles. Thus attention to this table is important when scheduling RSE instructions.

Using Table 4 we were able to reschedule instructions such that almost all RSE-related bubbles were eliminated, that is, all of the ones recorded by counters `be_exe_bubble.arcr` and `be_rse_bubble.ar_dep`, plus most of `be_rse_bubble.loadrs`. In total, 23 of the 25 RSE-related bubbles were eliminated, resulting in a total execution time of 72 cycles. The remaining 2 bubbles (from `loadrs` and `flushrs` instructions) are unavoidable (see Section 2.4.2).

System-instruction latencies Of the remaining 31 bubbles, 16 are due to counter `be_exe_bubble.grall`. These relate to general register scoreboard stalls, which in this case result from accesses to long-latency registers such as the *kernel register* that is used to hold the current thread ID. Hence we measured latencies of system instructions and registers. For this we used a modified Linux kernel, where we made use of gate pages to execute privileged instructions from a user-level program. The modified Linux kernel allows user-space code to create gate pages using `mprotect()`. Executing privileged instructions from user-level code greatly simplified taking the required measurements.

Our results are summarised in Table 5. Fortunately, register latencies are now provided in the latest version of the Itanium 2 Processor Reference Manual [9], so they are not included in this table. Unlike the RSE-induced latencies, our coverage of system-instruction latencies is presently still incomplete, but sufficient for the case at hand. Using this information we eliminated the 16 remaining execution-unit-related bubbles, by scheduling useful work instead of allowing the processor to stall.

Data-load stalls This leaves 15 bubbles due to data load pipeline stalls, counted as `be_l1d_fpu_bubble.l1d_dcurecir` and `be_l1d_fpu_bubble.l1d_stbufrecir`. The Itanium 2 Processor Reference Manual explains the former as “back-end was stalled by L1D due to DCU recirculating” and the latter as “back-end was stalled by L1D due to store buffer cancel needing recirculate”, which is hardly enlightening. We determined that the store buffer recirculation was most likely due to address conflicts between loads and stores (a load following a store to the same cache line within 3 cycles), due to the way we had scheduled loads and stores in parallel. Even

From	To	cyc	PMU counter
mov ar.rsc=	RSE_AR	13	BE_RSE_BUBBLE.AR_DEP
mov ar.bspstore=	RSE_AR	6	BE_RSE_BUBBLE.AR_DEP
mov =ar.bspstore	mov ar.rnat=	8	BE_EXE_BUBBLE.ARCR
mov =ar.bsp	mov ar.rnat=	8	BE_EXE_BUBBLE.ARCR
mov =ar.rnat/ar.unat	mov ar.rnat/ar.unat=	6	BE_EXE_BUBBLE.ARCR
mov ar.rnat/ar.unat=	mov =ar.rnat/ar.unat	6	BE_EXE_BUBBLE.ARCR
mov =ar.unat	FP_OP	6	BE_EXE_BUBBLE.ARCR
mov ar.bspstore=	flushrs	12	BE_RSE_BUBBLE.OVERFLOW
mov ar.rsc=	loadrs	12	BE_RSE_BUBBLE.LOADRS
mov ar.bspstore=	loadrs	12	BE_RSE_BUBBLE.LOADRS
mov =ar.bspstore	loadrs	2	BE_RSE_BUBBLE.LOADRS
loadrs	loadrs	8	BE_RSE_BUBBLE.LOADRS

Table 4: Experimentally-determined latencies for all combinations of two instructions involving the RSE. RSE_AR means any access to one of the registers ar.rsc, ar.bspstore, ar.bsp, or ar.rnat.

From	To	cyc	PMU counter
epc	ANY	1	-
bsw	ANY	6	BE_RSE_BUBBLE.BANK_SWITCH
rfi	ANY	13	BE_FLUSH_BUBBLE.BRU (1) , BE_FLUSH_BUBBLE.XPN (8) , BACK_END_BUBBLE.FE (3)
srlz.d	ANY	1	-
srlz.i	ANY	12	BE_FLUSH_BUBBLE.XPN (8) , BACK_END_BUBBLE.FE (3)
sum/rum/mov psr.um=	ANY	5	BE_EXE_BUBBLE.ARCR
sum/rum/mov psr.um=	srlz	10	BE_EXE_BUBBLE.ARCR
ssm/rsm/mov psr.l=	srlz	5	BE_EXE_BUBBLE.ARCR
mov =psr.um/psr	srlz	2	BE_EXE_BUBBLE.ARCR
mov pkr/rr=	srlz/sync/fwb/ mf/invalid_M0	14	BE_EXE_BUBBLE.ARCR
probe/tpa/tak/thash/ttag	USE	5	BE_EXE_BUBBLE.GRALL

Table 5: Experimentally-determined latencies for system instructions (incomplete). ANY means any instruction, while USE means any instruction consuming the result.

after eliminating this, there were still DCU recirculation stalls remaining.

While investigating this we noticed a few other undocumented features of the Itanium pipeline. It seems that most *application register* (AR) and *control register* (CR) accesses are issued to a limited-size buffer (of apparently 8 entries), with a “DCS stall” occurring when that buffer is full. No explanation of the acronym “DCS” is given in the Itanium manuals. It also seems that a DCU recirculation stall occurs if a DCS data return coincides with two L1 data-cache returns, which points to a limitation in the number of writeback ports. We also found that a DCU recirculation stall occurs if there is a load or store exactly 5 cycles after a move to a region register (RR) or protection-key register (PKR). These facts allowed us to identify the remaining stalls, but there may be other cases as well.

We also found a number of undocumented special split-issue cases. Split issue occurs after srlz, sync and mov =ar.unat and before mf instructions. It also occurs between a mov =ar.bsp and any B-unit instruction, as well as between an M-unit and an fwb instruction. There may be other cases.

We also found a case where the documentation on mapping of instruction templates to functional units is clearly incorrect. The manual says “ $M_A M_L I - M_S M_A I$ gets mapped to ports M2 M0 I0 – M3 M1 I1. If M_S is a getf instruction, a split issue will occur.” However, our experiments show that the mapping is really M1 M0 I0 – M2 M3 I1, and **no** split issue occurs in this case. It seems that in general the *load* subtype is allocated first.

Version	cycles	inst. grps	bubbles
C++ generic	508	231	277
Initial asm	170	83	87
Optimised	95	39	56
Final	36	33	3
Optimal	34	32	2
Archit. limit	9	9	0

Table 6: Comparison of IPC path optimisation, starting with the generic C++ implementation. *Optimised* refers to the version achieved using publicly available documentation, *final* denotes what was achieved after systematically measuring latencies. *Optimal* is what could be achieved on the present hardware with perfect instruction scheduling, while the *architectural limit* assumes unlimited resources and only single-cycle latencies.

5.3.4 Final optimisation

Armed with this knowledge we were able to eliminate all but one of the 15 data-load stalls, resulting in only 3 bubbles and a final execution time of 36 cycles, or 24ns on a 1.5GHz Itanium 2. This is extremely fast, in fact unrivalled on any other architecture. In terms of cycle times this is about a factor of two faster than the fastest RISC architecture (Alpha 21264) to which the kernel has been ported so far, and in terms of absolute time it is well beyond anything we have seen so far. This is a clear indication of the excellent performance potential of the Itanium architecture.

The achieved time of 36 cycles (including 3 bubbles) is actually still slightly short of the optimal solution on the present Itanium. The optimal solution can be found by examining the critical path of operations, which turns out to be 34 cycles (including 2 unavoidable bubbles for `flushrs` and `loadrs`). Significant manual rescheduling the code would (yet again) be necessary to achieve this 2 cycle improvement.

The bottlenecks preventing optimisation past 34 cycles are the kernel register read to obtain the current thread ID, which has a 12 cycle latency, and the latency of 12 cycles between `mov ar.bspstore=` (changing the RSE backing store pointer) and the following `loadrs` instruction. Also, since many of the instructions are system instructions which can only execute on a particular unit (M2), the availability of that unit becomes limiting. Additionally, it seems to be impossible to avoid a branch misprediction on return to user mode, as the predicted return address comes from the return stack buffer, but the nature of IPC is that it returns to a different thread. Eliminating those latencies would get us close to the architectural limit of Itanium, which is characterised as having unlimited resources (functional units) and only single-cycle latencies. This limit is a

mind-boggling 9 cycles! The achieved and theoretical execution times are summarised in Table 6.

The almost threefold speedup from 95 to 36 cycles made a significant difference for the performance of driver benchmarks within our component system. It would not have been possible without the powerful performance monitoring support on the Itanium processor, particularly the ability to break down stall events. The PMU allowed us to discover and explain all of the stalls involved.

This experience also helped us to appreciate the challenges facing compiler writers on Itanium. Without information such as that of Tables 4 and 5 it is impossible to generate truly efficient code. A compiler could use this information to drive its code optimisation, eliminating the need for labour-intensive hand-scheduled assembler code. Present compilers seem to be far away from being able to achieve this. While we have not analysed system-call code from other operating systems to the same degree, we would expect them to suffer from the same problems, and benefit from the same solutions. However, system-call performance is particularly critical in a microkernel, owing to the high frequency of kernel invocations.

6 Conclusion

As has been shown, the Itanium is a very interesting platform for systems programming. It presents a number of unusual features, such as its approach to address translation and memory protection, which are creating a new design space for systems builders.

The architecture provides plenty of challenges too, including managing its large register set efficiently, and overcoming hurdles to virtualisation. However, the most significant challenge of the architecture to systems implementors is the more mundane one of optimising the code. The EPIC approach has proven a formidable challenge to compiler writers, and almost five years after the architecture was first introduced, the quality of code produced by the available compilers is often very poor for systems code. Given this time scale, the situation is not likely to improve significantly for quite a number of years.

In the meantime, systems implementors who want to tap into the great performance potential of the architecture have to resort to hand-tuned assembler code, written with a thorough understanding of the architecture and its complex instruction scheduling rules. Performance improvements by factors of 2–3 are not unusual in this situation, and we have experienced cases where performance could be improved by an order of magnitude over GCC-generated code.

Such manual micro-optimisation is made harder by the unavailability of sufficiently detailed documentation.

This, at least, seems to be something the manufacturer should be able to resolve quickly.

Acknowledgements

This work was supported by a Linkage Grant from the Australian Research Council (ARC) and a grant from HP Company via the Gelato.org project, as well as hardware grants from HP and Intel. National ICT Australia is funded by the Australia Government's Department of Communications, Information Technology, and the Arts and the ARC through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

We would also like to thank UNSW Gelato staff Ian Wienand and Darren Williams for their help with benchmarking.

Notes

1. The results in [13] were obtained with kernels that were not fully functional and are thus somewhat optimistic. Also the processors used had shorter pipelines than modern high-end CPUs and hence lower hardware-dictated context switching costs. The figure of 70–150 cycles reflects (yet) unpublished measurements performed in our lab on optimised kernels for ARM, MIPS, Alpha and Power 4.

References

- [1] Aim benchmarks. <http://sourceforge.net/projects/aimbench>.
- [2] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proc. 1st OSDI*, pages 243–253, Monterey, CA, USA, 1994. USENIX/ACM/IEEE.
- [3] Matthew Chapman and Gernot Heiser. Implementing transparent shared memory on clusters using virtual machines. In *Proc. 2005 USENIX Techn. Conf.*, Anaheim, CA, USA, Apr 2005.
- [4] Matthew Chapman, Ian Wienand, and Gernot Heiser. Itanium page tables and TLB. Technical Report UNSW-CSE-TR-0307, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, May 2003.
- [5] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *Trans. Comp. Syst.*, 3:31–62, 1985.
- [6] HP Labs. *Perfmon*. <http://www.hpl.hp.com/research/linux/perfmon/>.
- [7] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, and Rumi Zahir. Introducing the IA-64 architecture. *IEEE Micro*, 20(5):12–23, 2000.
- [8] Intel Corp. *Itanium Software Conventions and Runtime Architecture Guide*, May 2001. <http://developer.intel.com/design/itanium/family>.
- [9] Intel Corp. *Intel Itanium 2 Processor Reference Manual*, May 2004. <http://developer.intel.com/design/itanium/family>.
- [10] Intel Corp. *Vanderpool Technology for the Intel Itanium Architecture (VT-i) Preliminary Specification*, Jan 2005. <http://www.intel.com/technology/vt/>.
- [11] L4Ka Team. L4Ka::Pistachio kernel. <http://l4ka.org/projects/pistachio/>.
- [12] Henry M. Levy and P. H. Lipman. Virtual memory management in the VAX/VMS operating system. *IEEE Comp.*, 15(3):35–41, Mar 1982.
- [13] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nay-eem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proc. 6th HotOS*, pages 28–31, Cape Cod, MA, USA, May 1997.
- [14] Daniel J. Magenheimer and Thomas W. Christian. vBlades: Optimised paravirtualisation for the Itanium processor family. In *Proc. 3rd Virtual Machine Research & Technology Symp.*, pages 73–82, 2004.
- [15] Larry McVoy and Carl Staelin. Imbench: Portable tools for performance analysis. In *Proc. 1996 USENIX Techn. Conf.*, San Diego, CA, USA, Jan 1996.
- [16] David Mosberger and Stéphane Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice Hall, 2002.
- [17] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *Proc. 5th OSDI*, Boston, MA, USA, Dec 2002.
- [18] Open Source Development Labs. *Database Test Suite*. http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite.
- [19] OpenGroup. The Single UNIX Specification version 3, IEEE std 1003.1-2001. http://www.unix-systems.org/single_unix_specification/, 2001.
- [20] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Comm. ACM*, 17(7):413–421, 1974.
- [21] Ryan Rakvic, Ed Grochowski, Bryan Black, Murali Annavaram, Trung Diep, and John P. Shen. Performance advantage of the register stack in Intel Itanium processors. In *2nd Workshop on EPIC Architectures and Compiler Technology*, Istanbul, Turkey, Nov 2002.
- [22] John W. Sias, Matthew C. Merten, Erik M. Nystrom, Ronald D. Barnes, Christopher J. Shannon, Joe D. Matarazzo, Shane Ryoo, Jeff V. Olivier, and Wen-mei Hwu. Itanium performance insights from the IMPACT compiler, Aug 2001.
- [23] SPARC International Inc., Menlo Park, CA, USA. *The SPARC Architecture Manual, Version 8*, 1991. <http://www.sparc.org/standards.html>.
- [24] John Wilkes and Bart Sears. A comparison of protection lookaside buffers and the PA-RISC protection architecture. Technical Report HPL-92-55, HP Labs, Palo Alto, CA, USA, Mar 1992.

Providing Dynamic Update in an Operating System

Andrew Baumann, Gernot Heiser

University of New South Wales & National ICT Australia

Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski

IBM T.J. Watson Research Center

Jeremy Kerr

IBM Linux Technology Center

Abstract

Dynamic update is a mechanism that allows software updates and patches to be applied to a running system without loss of service or down-time. Operating systems would benefit from dynamic update, but place unique demands on any implementation of such features. These demands stem from the event-driven nature of operating systems, from their restricted run-time execution environment, and from their role in simultaneously servicing multiple clients.

We have implemented a dynamic update mechanism in the K42 research operating system, and tested it using previous modifications of the system by kernel developers. Our system supports updates changing both kernel code and data structures. In this paper we identify requirements needed to provide a dynamically updatable operating system, describe our implementation, and present our experiences in designing and using the dynamic update mechanism. We also discuss its applicability to other operating systems.

1 Introduction

As computing infrastructure becomes more widespread, there has been an increasing number of patches for functionality, performance, and security reasons. To take effect, these patches traditionally require either restarting system services, or often rebooting the machine. This results in downtime. Sometimes this downtime can be scheduled, if for example the patch adds a feature, improves performance, etc. However, in some situations, such as applying a security patch, delaying the update is not desirable. Users and system administrators are forced to trade off the increased vulnerability of a security flaw against the cost of unplanned downtime.

Dynamic update [26] is used to avoid such downtime. It involves on-the-fly application of software updates to a running system without loss of service. The increased

unplanned down-time of computing infrastructure to apply updates, combined with the demand for continuous availability, provides strong motivation to investigate dynamic update techniques for operating systems.

In addition to the above mentioned impact on availability, dynamically updatable systems have other benefits. Such systems provide a good prototyping environment. They allow, for example, a new page replacement, file system, or network policy to be tested without rebooting. Further, in more mature systems such as mainframes, some user constraints prevent the system from ever being shutdown. In such an environment, users can only get new functionality into the system by performing a dynamic update.

An operating system is a unique environment with special constraints, and additional challenges must be solved to provide dynamic update functionality. We have addressed these challenges in the implementation of a dynamic update mechanism for K42, an object-oriented research operating system supporting hot-swapping. The focus of this paper is on the implementation and mechanisms needed to provide dynamic update. This work builds on previously reported work [6, 28], and on other K42 features. Some of the requisite characteristics we identify for dynamic update exist in other systems or have recently been incorporated [22], while others require additional support. Where appropriate, we point out the generality of our techniques to other operating systems, as well as what infrastructure would be required to take full advantage of dynamic update techniques. In addition to describing our implementation, we describe our experiences applying dynamic update in K42, using three motivating examples taken from changes made by K42 kernel developers.

The rest of this paper is organised as follows: Section 2 discusses the system requirements for supporting dynamic update, Section 3 describes our implementation of dynamic update in K42, and Section 4 discusses how the same functionality might be implemented in other op-

erating systems. Next, Section 5 describes our experiences applying dynamic update to K42 using three motivating examples, Section 6 discusses the limitations of our implementation and our plans for future work, Section 7 compares related work, and Section 8 concludes.

2 Requirements for dynamic update

There are several fundamental requirements in providing a dynamic update capability. Here we identify them, in Section 3.2 we describe how we satisfy them in K42, and then in Section 4 we generalise to other operating systems.

2.1 Classification of updates

At a minimum, dynamic update needs to support changes to the code of a system, however there are varying levels of support possible for updates which also affect data. We classify dynamic updates in this way:

1. Updates that only affect code, where any data structures remain unchanged across the update. This is easier to implement, but imposes significant limitations on what updates may be applied dynamically.
2. Updates that affect both code and global, single-instance, data. Examples of this might include changes to the Linux kernel's unified page cache structure, or to K42's kernel memory allocator.
3. Updates that affect multiple-instance data structures, such as the data associated with an open socket in Linux, or an open file.

2.2 Requirements

Having classified the possible updates, we now introduce a set of fundamental requirements for dynamic update.

Updatable unit: In order to update a system, it is necessary to be able to define an updatable unit. Depending on the class of update supported, and the implementation of the system, a unit may consist of a code module, or of both code and encapsulated data. In both cases, there must be a clearly defined interface to the unit. Furthermore, external code should invoke the unit in a well-defined manner, and should not arbitrarily access code or data of that unit.

While creating updatable units is easier with support from languages such as C++, it is still possible without such support. Primarily, providing updatable units means designing with good modularity and obeying module boundaries. The structure of the system dictates what is feasible.

Safe point: Dynamic updates should not occur while any affected code or data is being accessed. Doing so could cause undefined behaviour. It is therefore important to determine when an update may safely be applied. In general however, this is undecidable [15]. Thus, system support is required to achieve and detect a safe point. Potential solutions involve requiring the system to be programmed with explicit update points, or blocking accesses to a unit, and detecting when it becomes idle, or *quiescent*.

An operating system is fundamentally event-driven, responding to application requests and hardware events, unlike most applications, which are structured as one or more threads of execution. As discussed later, this event-based model can be used to detect when an updatable unit of the system has reached a safe point. Additional techniques can be employed to handle blocking I/O events or long running daemon threads.

State tracking: For a dynamic update system to support changes to data structures, it must be able to locate and convert all such structures. This requires identifying and managing all instances of state maintained by a unit in a uniform fashion, functionality usually provided in software systems using the factory design pattern [12]. Note that the first two classes of update, dynamic update to code and dynamic update to single-instance data, are still possible without factories, but it is not possible to support dynamic update affecting multiple-instance data without some kind of state tracking mechanism.

State transfer: When an update is applied affecting data structures, or when an updated unit maintains internal state, the state must be transferred, so that the updated unit can continue transparently from the unit it replaced. The state transfer mechanism performs this task, and is how changes to data structures can be supported.

Redirection of invocations: After the update occurs, all future requests affecting the old unit should be redirected. This includes invocations of code in the unit. Furthermore, in a system supporting multiple-instance data structures, creation of new data structures of the affected type should produce the updated data structure.

Version management: In order to package and apply an update, and in order to debug and understand the running system, it is necessary to know what code is actually executing. If an update depends on another update having previously been applied, then support is required to be able to verify this. Furthermore, if updates are from multiple sources, the versioning may not be linear, caus-

ing the interdependencies between updates to become complex and difficult to track.

The level of support required for version management is affected by the complexity of update interdependencies, but at a minimum it should be possible to track a version number for each update present in the system, and for these version numbers to be checked before an update is applied.

3 Dynamic update in K42

We now describe our implementation of dynamic update in K42. As noted previously, some of the techniques used in the implementation are specific to K42, but other operating systems are becoming more amenable to dynamic update, as discussed in the next section.

3.1 K42

The K42 project is developing a new scalable open-source research operating system incorporating innovative mechanisms and policies, and modern programming technologies. It runs on 64-bit cache-coherent PowerPC systems, and supports the Linux API and ABI. It uses a modular object-oriented design to achieve multiprocessor scalability, enhance customisability, and enable rapid prototyping of experimental features (such as dynamic update).

Object-oriented technology has been used throughout the system. Each resource (for example, virtual memory region, network connection, open file, or process) is managed by a different set of object instances [5]. Each object encapsulates the meta-data necessary to manage the resource as well as the locks necessary to manipulate the meta-data, thus avoiding global locks, data structures, and policies. The object-oriented nature enables adaptability, because different resources can be managed by different implementations. For example, each running process in the system is represented by an in-kernel instance of the *Process* object (analogous to the process control block structure present in other operating systems). Presently two implementations of the *Process* interface exist, *ProcessReplicated*, the default, and *ProcessShared*, which is optimised for the case when a process exists on only a single CPU [2]. The K42 kernel defaults to creating replicated processes, but allows for a combination of replicated and shared processes.

K42 uses clustered objects [4], a mechanism that enables a given object to control its own distribution across processors. Using the object translation table facility provided by clustered objects, hot-swapping [4, 28] was implemented in K42. Hot-swapping allows an object instance to be transparently switched to another implemen-

tation while the system is running, and forms the basis of our dynamic update implementation.

3.2 Support for dynamic update

Requirements

In Section 2.2, we identified several requirements for dynamic update of an operating system. In K42, these requirements are addressed by our implementation of the dynamic update mechanism, as follows:

Updatable unit: A good choice for the dynamically updatable unit in K42 is the same as for hot-swapping, namely the object instance. K42 is structured as a set of objects, and the coding style used enforces encapsulation of data within objects. Each object's interface is declared in a virtual base class, allowing clients of an object to use any implementation, and for the implementation to be changed transparently by hot-swapping.

Safe point: K42 detects quiescent states using a mechanism similar to read copy update (RCU) in Linux [22, 23]. This technique makes use of the fact that each system request is serviced by a new kernel thread, and that all kernel threads are short-lived and non-blocking.

Each thread in K42 belongs to a certain epoch, or *generation*, which was the active generation when it was created. A count is maintained of the number of live threads in each generation, and by advancing the generation and waiting for the previous generations' counters to reach zero, it is possible to determine when all threads that existed on a processor at a specific instance in time have terminated [13].

The implementation blocks new invocations of an object being updated, and then uses the generation-count mechanism to detect quiescence [28].

State tracking: state-tracking is provided by factory objects, which are described in detail in Section 3.3.

State transfer: Once the object being swapped is quiescent, the update framework invokes a state transfer mechanism which transfers state from the old object to the new object, using a *transfer negotiation protocol* to allow the negotiation of a common intermediate format that both objects support [28]. Object developers must implement data conversion functions to and from common intermediate formats.

This generalised technique was developed to support hot-swaps between arbitrary implementations of an object. In the case of dynamic update, usually the replacement object is merely a slightly modified version of the original object, with similar state information, so the

conversion functions perform either a direct copy, or a copy with slight modifications.

In cases where a lot of state is maintained, or when many object instances must be updated, a copy is an unnecessary expense, because the updated object is deleted immediately afterwards. For example, the process object maintains a series of structures which describe the process' address space layout. To avoid the cost of deep-copying and then discarding these structures, the data transfer functions involved simply copy the pointer to the structure and set a flag in the old object. When the object is destroyed it checks this flag and, if it is set, does not attempt destruction of the transferred data structures. Effectively ownership of the structure is transferred to the new object instance. This only works in cases where the new object uses the same internal data format as the old object. This is true in many dynamic update situations. In cases where this is not true, the negotiation protocol ensures that a different transfer function is used.

Redirection of invocations: K42 uses a per-address-space *object translation table*. Each object has an entry in the table, and all object invocations are made through this reference. In the process of performing a dynamic update, the translation table entries for an object are updated to point to the new instance, which causes future calls from clients to transparently invoke the new code. The object translation table was originally introduced into K42 to support the clustered object multiprocessor scalability mechanism [13], and we have been able to utilise it to implement hot-swapping and thus dynamic update.

When an object that has multiple instances is updated, we must also redirect creations of that type. This redirection is provided by the factory mechanism, described in Section 3.3.

Version management: We have implemented a simple versioning scheme for dynamic updates in K42. Each factory object carries a version number, and before an update proceeds these version numbers are checked. Further details follow in Section 3.3.

Hot-swapping

Because hot-swapping forms the basis of dynamic update, we outline its implementation here. Further details are available in previous papers [4,28].

As we have mentioned, the object translation table adds an extra level of indirection on all object invocations. This indirection enables an interposition mechanism whereby an object's entry in the object translation table is modified, causing all accesses to that object to transparently invoke a different interposer object. The

interposer can then choose to pass the call along to the original object. This mechanism is used by the hot swapping and dynamic update implementations.

Hot-swapping operates by interposing a mediator object in front of the object to be hot-swapped. The mediator passes through several phases, first tracking incoming calls until it knows (through the generation-count mechanism) that all calls are being tracked, then suspending further calls until the existing tracked calls complete. At this point the object is quiescent. The mediator then performs state transfer format negotiation, followed by the state transfer between the old and the new object instances. Finally, it updates the object translation table reference to the new object, and forwards the blocked calls.

3.3 Dynamic update implementation

Module loader

To perform updates, the code for the updated object must be present. The normal process for adding an object to K42 was to recompile the kernel, incorporating the new object, and then reboot the system. This is insufficient for dynamic update, so we have developed a *kernel module loader* that is able to load the necessary code for an updated object into a running kernel or system server.

A K42 kernel module is a relocatable ELF file with unresolved references to standard kernel symbols and library routines (such as *err_printf*, the console output routine). Our module loader consists of a special object in the kernel that allocates pinned memory in the kernel's text area, and a trusted user-space program that has access to the kernel's symbol table. This program uses that symbol table to resolve the undefined symbols in the module, and load it into the special region of memory provided by the kernel object. It then instructs the kernel to execute the module's initialisation code.

Our module loader operates similarly to that used in Linux [8], but is simpler. Linux must maintain a dynamic symbol table and support interdependencies between modules, we avoid this because all objects are invoked indirectly through the object translation tables. A module can (and to be useful should) contain code that is called by the existing kernel without requiring its symbols to be visible. Its initialisation code simply instantiates replacement objects and performs hot-swap operations to invoke the code in those object instances. Our module loader performs the relocations and symbol table management at user-level, leaving only the space allocator object in the kernel.

Factory mechanism

Hot-swapping allows us to update the code and data of a single specific object instance. However, K42 is structured such that each instance of a resource is managed by a different instance of an object. To dynamically update a kernel object, the infrastructure must be able to both locate and hot-swap all instances of that object, and cause any new instantiations to use the updated object code. Note that, as we have mentioned, this problem is not unique to K42; to support dynamic updates affecting data structures requires a mechanism to track all instances of those data structures and update them.

Previously in K42, object instances were tracked in a class-specific manner, and objects were usually created through calls to statically-bound methods. For example, to create an instance of the *ProcessReplicated* object (the implementation used by default for *Process* objects), the call used was:

```
ProcessReplicated::Create(  
    ProcessRef &out, HATRef h, PMRef pm,  
    ProcessRef creator, const char *name);
```

This leads to problems for dynamic update, because the *Create* call is bound at compile-time, and cannot easily be redirected to an updated implementation of the *ProcessReplicated* object, and also because we rely on the caller of this method to track the newly created instance.

To track object instances and control object instantiations, we used the factory design pattern [12]. In this design pattern, the factory method is an abstraction for creating object instances. In K42, factories also track instances that they have created, and are themselves objects. Each factory object provides an interface for creating and destroying objects of one particular class, and maintains the set of objects that it has created.

The majority of the factory implementation is factored out using inheritance and preprocessor macros, so that adding factory support to a class is relatively simple. Using our previous example, after adding the factory, the creation call changed to:

```
DREF_FACTORY_DEFAULT(ProcessReplicated)  
->create(...);
```

where (...) represents the arguments as before. The macro above hides some implementation details, whereby the default factory for a class is referenced using a static member; it expands to the following:

```
(*ProcessReplicated::Factory::factoryRef)  
->create(...);
```

Using a factory reference allows us to hot-swap the factory itself, which is used in our implementation of dynamic update.

To provide rudimentary support for configuration management, factories carry a version number identifying the specific implementation of the factory and its type. The factories in the base system all carry version zero, and updated factories have unique non-zero version numbers. We assume a strictly linear model of update, when an update occurs the current version number of the factory is compared to the version number of the update, and if the update is not the immediately succeeding version number, the update is aborted. To support reverting updates in this scheme, we could reapply the previous version with an increased version number.

Performance and scalability influenced our implementation of the factories. For example, object instances are tracked for dynamic update in a distributed fashion using per-CPU instance lists. Moreover, we found that adding factories to K42 was a natural extension of the object model, and led to other advantages besides dynamic update. As an example, in order to choose between *ProcessReplicated* and *ProcessShared*, K42 had been using a configuration flag that was consulted by the code that creates process objects to determine which implementation to use. Using the factory model, we could remove this flag and allow the scheme to support an arbitrary number of implementations, by changing the default process factory reference to the appropriate factory object.

Steps in a dynamic update

We use factories to implement dynamic update in K42. To perform a dynamic update of a class, the code for the update is compiled along with some initialisation code into a loadable module. When the module is loaded, its initialisation code is executed. This code performs the following steps (illustrated in Figure 1):

1. A factory for the updated class is instantiated. At this point the version number of the updated factory is checked against the version number of the existing factory, if it is incorrect the update is aborted.
2. The old factory object is located using its statically bound reference, and hot-swapped to the new factory object; during this process the new factory receives the set of instances that was being maintained by the old factory.
3. Once the factory hot-swap has completed, all new object instantiations are being handled by the new updated factory, and therefore go to the updated class. However, any old instances of the object have not yet been updated.
4. To update the old instances, the new factory traverses the set of instances it received from the old factory. For each old instance it creates an instance

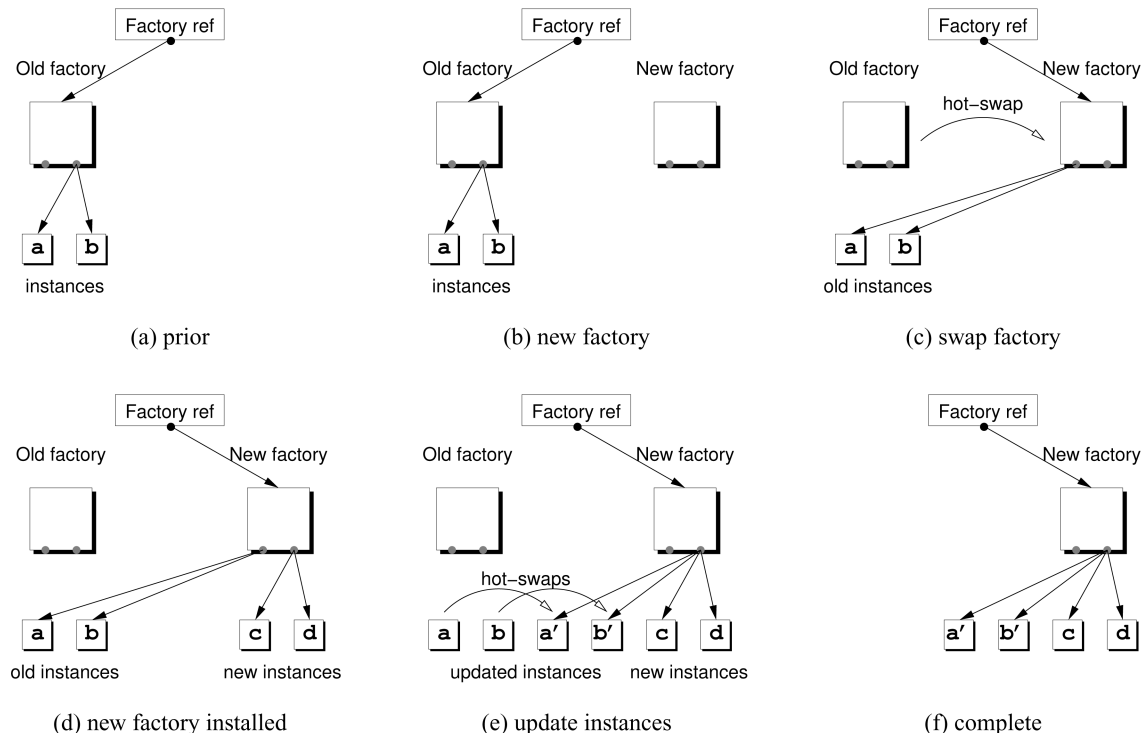


Figure 1: Phases in the dynamic update of a multiple-instance object using a factory: (a) prior to update the old factory is maintaining instances of a class; (b) instantiate a new factory for the updated class; (c) hot-swap the old factory with its new replacement (transferring the list of managed instances); (d) after the hot-swap completes, new instantiations are handled by the updated factory code (thus creating objects of the new type); (e) update old instances by traversing the list and hot-swapping each instance to an updated replacement (occurs in parallel on each CPU); (f) the update is complete.

of the updated object, and initiates a hot-swap between the old and the new instances.

This step proceeds in parallel across all CPUs where the old factory was in use, and while the rest of the system is functioning. Because each object instance is hot-swapped individually, and because K42 encapsulates all data behind object instances, there is no requirement to block all accesses to all objects of the affected type while an update is in progress.

5. Finally, the update is complete and the old factory is destroyed.

In some special cases, for example when an update adds new objects to the system that do not replace any existing objects, or when an update affects an object with only a single instance, the full dynamic update implementation is not required. In these cases the initialisation code in the module is simpler.

4 Dynamic update in other systems

In this section we discuss the way in which a dynamic update mechanism might be provided in operating systems other than K42, focusing on Linux. Previously we identified several key requirements, these might be provided as follows:

Updatable unit: Modularity is already widely used in constructing operating systems, the virtual file system (VFS) layer [20] being one well-known example. Furthermore, modern operating systems are being constructed in an increasingly modular fashion to allow for better multiprocessor performance, and improved customisability. This trend toward modularity provides the necessary interfaces for creating updatable units. Even monolithic systems such as Linux now have support for loadable kernel modules for device drivers, file systems, networking functionality, etc. This support could be leveraged to provide dynamic update capabilities in the same areas where kernel modules can be used.

Safe point: Linux has recently incorporated the quiescence detection mechanism known as RCU [22], which is similar to the generation count mechanism used in K42. We expect that other operating systems would also be able to add RCU-style quiescence detection, which offers other benefits, such as improved scalability.

State tracking: A state tracking mechanism, such as a factory, is needed when dynamic update supports changes to the format of multiple-instance data, to locate all instances of that data. The factory design pattern is a generally well-understood software construction technique, and we would expect that factories could be added to an existing system. For example, in the Linux kernel, modules already maintain reference count and dependency information to prevent them from being unloaded while in use [25]. If the addition of factories was required, the modules could also be made responsible for tracking instances of their own state.

State transfer: The implementation of state transfer is something fairly unique to hot-swapping and dynamic update. In a system with clearly defined updatable units, it should be straightforward to implement the equivalent of K42's transfer negotiation protocol and state transfer functions.

Redirection of invocations: Few systems include a uniform indirection layer equivalent to K42's object translation table. Many systems such as Linux do use indirection to implement device driver abstractions or the VFS layer, and these pointers could be used to implement dynamic update. However, the lack of a uniform indirection mechanism would limit the applicability of dynamic update to those specific areas of the system.

For dynamic update to multiple-instance data structures to be supported, it is desirable that each instance be individually updatable. For example, the VFS layer's use of one set of function pointers per node, rather than one set for the entire file system, allows the file system's data structures to be converted incrementally. The alternative would be to block all access to all file system nodes while they are updated, effectively halting the whole system.

Version management: Versioning is an open problem for dynamic update. If our simple model of factory version numbers proves sufficient, it can be implemented in other systems.

Beyond these requirements, the dynamic update implementation also relies on a module loader to make the code for the update available in the running system.

Loadable modules are already widely used, most operating systems include a kernel module loader or equivalent functionality, so this is not a significant constraint.

5 Experiments

5.1 Performance measurements

We have performed a number of measurements to evaluate the performance penalty imposed by our dynamic update mechanism. All the experiments were conducted on an IBM pSeries 630 Model 6E4 system, with four 1.2GHz POWER4+ processors and 8GB of main memory.

Overhead of factory mechanism

We ran microbenchmarks to directly measure the code of adding the factory mechanism. Using a factory for an object implies extra cost when creating the object, because the creation must be recorded in the factory's data structures.

We measured the cost of creating three different objects using a factory and using a statically bound method, this includes allocating storage space for the object, instantiating it, and invoking any initialisation methods. Each test was repeated 10,000 times, and the total time measured using the processor's cycle counter. Our results are summarised in Table 1.

The first object, a dummy object, was created specifically for this test, and encapsulates a single integer value. This result represents the worst-case for adding a factory, with an overhead of 12% over the base creation cost of 2.22 μ s. The next object is an FCM (file cache manager), an instance of which is maintained by K42 for every open file in the system. Creating an FCM object is 5.6% slower with a factory, but this number overstates the true impact, because in practice FCM creation is usually followed by a page fault, which must be satisfied either by zero-filling a page, or by loading it from disk. Finally, we measured the cost of creating a process object, a relatively expensive operation because it involves initialising many other data structures, and found that in such a case, the additional cost imposed by the factory was very small, even before considering the additional costs involved in process creation such as context switches and initial page faults.

To get a more complete picture of the overhead imposed by the presence of factories, we used the SPEC software development environment throughput (SDET) benchmark [11]. This benchmark executes one or more scripts of user commands designed to simulate a typical software-development environment (for example, editing, compiling, and various UNIX utilities). The scripts

<i>object</i>	<i>static create</i>	<i>factory create</i>	<i>overhead</i>
dummy	2.22 μ s	2.49 μ s	12%
file	4.37 μ s	4.61 μ s	5.6%
process	61.1 μ s	61.5 μ s	0.73%

Table 1: Cost of creating various kernel objects with and without a factory.

are generated from a predetermined mix of commands, and are all executed concurrently. It makes extensive use of the file system, process, and memory management subsystems.

We ran SDET benchmarks with four configurations of the system:

1. the base system with no factories in use,
2. a factory on FCM objects,
3. a factory on process objects,
4. factories on both FCM and process objects.

We found that in these cases the use of factories imposes no noticeable performance degradation on system throughput, as measured by SDET. We have not yet extended factories to other objects in the system, however we expect the performance impact to also be minor, since FCMs and processes constitute a significant portion of the kernel objects created.

Time to perform an update

The cost of performing a dynamic update itself is more significant, in some initial experiments we measured 20ms to update one hundred live instances of the dummy object. This is because the hot-swapping implementation is not yet optimised for the case where large numbers of objects are swapped concurrently. We plan to improve this, although since a dynamic update does not block the entire system while it is applied, the overall time taken for an update is less critical.

5.2 Experiences applying dynamic update

To demonstrate the effectiveness of dynamic update, we present three examples of system changes of increasing complexity that we have applied using dynamic update. These examples relate to the memory management code of the K42 kernel [3, 19].

New kernel interfaces for partitioned memory region

Benchmarking of a memory-intensive parallel application showed poor scalability during its initialisation

phase. Analysis of the problem determined that a bottleneck occurred during the resizing of a shared hash table structure, and a new partitioned memory management object was developed that did not suffer from the problem. This object added a new interface to the kernel, allowing user programs to create a partitioned memory region if they specified extra parameters.

Adding a new piece of code to the kernel and making it available through a new interface is the simplest case of dynamic update, because we can avoid replacing old code or transferring state information. This update was implemented as a simple loadable module, consisting of the code for the new region object and some initialisation code to load it and make it available to user programs. This module could be shipped with programs requiring it, or could be loaded into the kernel on demand when a program requires the new interface, either way avoiding a reboot.

This scenario demonstrates the use of a module loader combined with an extensible kernel interface to add new functionality to a running kernel. There is nothing inherently new here, existing systems also allow modules to be loaded, for example to provide new file systems or device drivers. However, K42's modularity makes it possible to replace a portion of the page-fault path for a critical application with a new set of requirements, which could not be done on an existing system such as Linux.

Fix for memory allocator race condition

This scenario involves a bug fix to a kernel service, one of the key motivations for dynamic update. In the course of development, we discovered a race condition in our core kernel memory allocator that could result in a system crash when kernel memory was allocated concurrently on multiple CPUs.

Fixing this bug required adding a lock to guard the allocation of memory descriptors, a relatively simple code change. In fact, only two lines of code were added, one to declare the lock data structure, and another to acquire (and automatically release on function return) the lock. A recompile and reboot would have brought the fix into use. However, even with continual memory allocation and deallocation occurring, we were able to avoid the reboot and dynamically apply the update using our mechanism.

The replacement code was developed as a new class inheriting almost all of its implementation from the old buggy object, except for the declaration of the lock and a change to the function that acquired and released it. This caused the C++ compiler to include references to all the unchanged parts of the old class in the replacement object code, avoiding programmer errors. Simple copying implementations of the state transfer functions were also

provided to allow the object to be hot-swapped. The key parts of the replacement code are shown in Figure 2.

The new class was compiled into a loadable module, and combined with initialisation code that instantiated the new object and initiated a hot-swap operation to replace the old, buggy instance. Because this object was a special case object with only a single instance, it was not necessary to use the factory mechanism.

This scenario demonstrates the use of hot-swapping as part of our dynamic update mechanism, combined with a kernel module loader, to dynamically update live code in the system.

File cache manager optimisation

This scenario involves an optimisation to the implementation of file cache manager objects. An FCM is instantiated in the K42 kernel for each open file or memory object in the system.

We discovered that the *unmapPage* method did not check if the page in question was already unmapped before performing expensive synchronisation and IPC operations. These were unnecessary in some cases.

We developed a new version of the FCM object that performed the check before unmapping, and prepared it as a loadable module. Applying this update dynamically required the factory mechanism, because the running system had FCM instances present that needed to be updated, and because new instantiations of the FCM needed to use the code in the updated module.

This scenario demonstrates all the components of our dynamic update implementation, using a module loader to load the code into the system, a factory to track all instances of state information that are affected by an update, and hot-swapping to update each instance.

6 Open issues

The implementation we have accomplished thus far provides a basic framework for performing dynamic update. This is a rich area for investigation, and is becoming important for providing usable and maintainable systems. Here we discuss areas for future work.

Changing object interfaces: Due to a limitation of the current hot-swapping implementation, and because there is no support for coordinated swapping of multiple inter-dependant objects, we cannot dynamically apply updates that change object interfaces. We could enable this by extending the hot-swapping mechanism to support simultaneous swaps of multiple objects, namely the object whose interface is to be changed and all objects possibly using that interface.

This is our next step in expanding the effectiveness of dynamic update. When considering various changes to K42 to use as example scenarios for this work, many had to be rejected because they involved interface changes. While such changes might be less common in a production system, rather than a rapidly evolving experimental system such as K42, the restriction on changing object interfaces is currently the most serious limitation of this work.

Updates to low-level exception code: Another open issue is what code can be dynamically upgraded. Currently our mechanism requires the extra level of indirection provided by the object translation table for tracking and redirection. Low-level exception handling code in the kernel is not accessed via this table, and as such can not currently be hot-swapped or dynamically updated. It may be possible to apply some dynamic updates through the indirection available in the exception vectors, or through careful application of binary rewriting (for example, changing the target of a branch instruction), but it is difficult to envision a general-purpose update mechanism for this code. There is an open issue for both K42 and other operating systems should we desire the ability to update such low-level code.

Updates affecting multiple address spaces: Our update system does not yet support updates to code outside the kernel, such as system libraries or middleware. At present, it is possible to perform an update in an application's address space, but there is no central service to apply an update to all processes which require it. We intend to develop operating system support for dynamically updating libraries in a coordinated fashion. As part of this work we may need to extend the factory concept to multiple address spaces. We also need to consider the possible implications of changing cross-address-space interfaces, such as IPC interactions or the system call layer.

Timeliness of security updates: For some updates, such as security fixes, it is important to know when an update has completed, and to be able to guarantee that an update will complete within a certain time frame. It may be possible to relax the timeliness requirement by applying updates lazily, marking objects to be updated and only updating them when they are actually invoked, as long as we can guarantee that the old code will not execute once an object has been marked for update.

State transfer functions: State transfer between the old and new versions of an object is performed by the hot-swap mechanism using state transfer methods: the

```

class PageAllocatorKernPinned_Update : public PageAllocatorKernPinned {
public:
    BLock nextMemDescLock;

    void pinnedInit(VPNum numaNode) {
        nextMemDescLock.init();
        PageAllocatorKernPinned::pinnedInit(numaNode);
    }

    virtual void* getNextForMDH(uval size) {
        AutoLock<BLock> al(&nextMemDescLock); // locks now, unlocks on return
        return PageAllocatorKernPinned::getNextForMDH(size);
    }

    DEFINE_GLOBALPADDED_NEW(PageAllocatorKernPinned_Update); // K42-ism
};

```

Figure 2: Update source code for second example scenario (memory allocator race condition).

old object provides a method to export its state in a standard format, which can be read by the new object's import method. This works well enough, but it requires the tedious implementation of the transfer code, even though most updates only make minor changes, if any, to the instance data (for example, adding a new data member). It may be possible to partially automate the creation of state transfer methods in such cases, as has been done in other dynamic update systems [17,21].

An alternative approach to this problem is used by Nooks to support recoverable device drivers in Linux [31]. In this system, shadow drivers monitor all calls made into and out of a device driver, and reconstruct a driver's state after a crash using the driver's public API. Only one shadow driver must be implemented for each device class (such as network, block, or sound devices), rather than for each driver. A similar system could be used for dynamic update, instead of relying on objects to provide conversion functions, a shadow object could monitor calls into each updatable kernel object, reconstructing the object's state after it is updated. This approach suffers from several drawbacks. First, there is a continual runtime performance cost imposed by the use of shadows, unlike conversion functions which are only invoked at update time. Secondly, the use of shadow drivers is feasible because there is a small number of device interfaces relative to the number of device drivers, but this is generally not the case for arbitrary kernel objects, which implement many different interfaces.

Failure recovery: We do not currently handle all the possible failures that could occur during an update. While we cannot detect arbitrary bugs in update code, it is possible for an update to fail in a recoverable man-

ner. For example, if the state transfer functions return an error code, the update should be aborted. Furthermore, resource constraints may prevent an update from being applied, because during an update both old and new versions of the affected object co-exist, consuming more memory. The system should either be able to check that an update can complete before attempting to apply it, or support transactions [27] to roll back a failed update.

Update preparation from source code: We need a mechanism to automate the preparation of updates from source code modifications. This could possibly be driven by make, using a rebuild of the system and a comparison of changed object files to determine what must be updated. However, it would be extremely difficult to build a completely generic update preparation tool, because changes to the source code of an operating system can have far-reaching and unpredictable consequences.

Configuration management: Our simple update versioning scheme assumes a linear update model, each update to a given class depends on all previous updates having been applied before it. Most dynamic update systems that have automated the update preparation and application process, have also assumed a linear model of update [17,21]. This is most likely inadequate for real use, where updates may be issued by multiple sources, and may have complex interdependencies.

More complex versioning schemes exist, such as in the .NET framework, where each assembly carries a four-part version number, and multiple versions may coexist [24]. We will need to reconsider versioning issues once we start automating the update preparation process, and more developers start using dynamic updates.

7 Related work

Previous work with K42 developed the hot-swapping feature [28], including the quiescence detection, state transfer, and object translation table mechanisms. Our work extends hot-swapping by adding the state tracking, module loader, and version management mechanisms, and combining them to implement dynamic update.

To our knowledge, no other work has focused on dynamic update in the context of an operating system. Many systems for dynamic updating have been designed, and a comprehensive overview of the field is given by Segal and Frieder [26]. These existing systems are generally either domain-specific [9, 10, 16], or rely on specialised programming languages [1, 17, 21], making them unsuitable for use in an operating system implemented in C or C++.

Proteus [29] is a formal model for dynamic update in C-like languages. Unlike our system for achieving quiescence on a module level, it uses pre-computed safe update points present in the code. Our system can also support explicit update points, however we have not found this necessary due to the event-driven programming used in K42.

Dynamic C++ classes [18] may be applicable to an updatable operating system. In this work, automatically-generated proxy classes are used to allow the update of code in a running system. However, when an update occurs it only affects new object instantiations, there is no support for updating existing object instances, which is important in situations such as security fixes. Our system also updates existing instances, using the hot-swapping mechanism to transfer their data to a new object.

Some commercial operating systems offer features similar to Solaris Live Upgrade [30], which allows changes to be made and tested without affecting the running system, but requires a reboot for changes to take effect.

Component- and microkernel-based operating systems, where services may be updated and restarted without a reboot, also offer improved availability. However, while a service is being restarted it is unavailable to clients, unlike our system where clients perceive no loss of service. Going a step further, DAS [14] supported dynamic update through special kernel primitives, although the kernel was itself not updatable.

Finally, extensible operating systems [7, 27] could potentially be modified to support dynamic update, although updates would be limited to those parts of the system with hooks for extensibility, and many of the same problems addressed here (such as state transfer) would be encountered.

8 Conclusion

We have presented a dynamic update mechanism that allows patches and updates to be applied to an operating system without loss of the service provided by the code being updated. We outlined the fundamental requirements for enabling dynamic update, presented our implementation of dynamic update, and described our experiences applying several example dynamic updates to objects taken from changes made by kernel developers. Our implementation also incurs negligible performance impact on the base system.

Although dynamic update imposes a set of requirements on operating systems, as described in this paper, those requirements are already being incrementally incorporated into systems such as Linux. This includes RCU to detect quiescent points, more modular encapsulated data structures, and the indirection needed to redirect invocations. We expect that dynamic update will become increasingly important for mainstream operating systems.

Acknowledgements

We wish to thank the K42 team at IBM Research for their input, in particular Marc Auslander and Michal Ostrowski, who contributed to the design of the kernel module loader, and Bryan Rosenburg, whose assistance in debugging K42 was invaluable. We also thank Raymond Fingas, Kevin Hui, and Craig Soules for their contributions to the underlying hot-swapping and interposition mechanisms, and finally our paper shepherd, Vivek Pai.

This work was partially supported by DARPA under contract NBCH30390004, and by a hardware grant from IBM. National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

Availability

K42 is released as open source and is available from a public CVS repository; for details refer to the K42 web site: <http://www.research.ibm.com/K42/>.

References

- [1] Jakob R. Andersen, Lars Bak, Steffen Garup, Kasper V. Lund, Toke Eskildsen, Klaus Marius Hansen, and Mads Torgersen. Design, implementation, and evaluation of the Resilient Smalltalk embedded platform. In *Proceedings of the 12th Eu-*

ropean Smalltalk User Group Conference, Köthen, Germany, September 2004.

- [2] Jonathan Appavoo, Marc Auslander, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenberg, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, Ben Gamsa, Reza Azimi, Raymond Fingas, Adrian Tam, and David Tam. Enabling scalable performance for general purpose workloads on shared memory multiprocessors. IBM Research Report RC22863, IBM Research, July 2003.
- [3] Jonathan Appavoo, Marc Auslander, David Edelson, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenberg, Robert W. Wisniewski, and Jimi Xenidis. Providing a Linux API on the scalable K42 kernel. In *Proceedings of the 2003 USENIX Technical Conference, FREENIX Track*, pages 323–336, San Antonio, TX, USA, June 2003.
- [4] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Craig A. N. Soules. An infrastructure for multiprocessor run-time adaptation. In *Proceedings of the ACM SIGSOFT Workshop on Self-Healing Systems*, pages 3–8, Charleston, SC, USA, November 2002.
- [5] Marc Auslander, Hubertus Franke, Ben Gamsa, Orran Krieger, and Michael Stumm. Customization lite. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, May 1997.
- [6] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Improving operating system availability with dynamic update. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, USA, October 2004.
- [7] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 267–284, Copper Mountain, CO, USA, December 1995.
- [8] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2nd edition, 2002.
- [9] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *Proceedings of the ACM Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 403–417, Anaheim, CA, USA, October 2003.
- [10] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd ICSE*, pages 470–476, San Francisco, CA, USA, 1976.
- [11] Steven L. Gaede. Perspectives on the SPEC SDET benchmark. Available from <http://www.specbench.org/osg/sdm91/sdet/>, January 1999.
- [12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- [13] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, New Orleans, LA, USA, February 1999. USENIX.
- [14] Hannes Goullon, Rainer Isle, and Klaus-Peter Löhr. Dynamic restructuring in an experimental operating system. In *Proceedings of the 3rd ICSE*, pages 295–304, Atlanta, GA, USA, 1978.
- [15] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996.
- [16] Steffen Hauptmann and Josef Wasel. On-line maintenance with on-the-fly software replacement. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 70–80, Annapolis, MD, USA, May 1996. IEEE Computer Society Press.
- [17] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23. ACM, June 2001.
- [18] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ classes—a lightweight mechanism to update code in a running program. In *Proceedings of the 1998 USENIX Technical Conference*, pages 65–76, June 1998.
- [19] IBM K42 Team. *Memory Management in K42*, August 2002. Available from <http://www.research.ibm.com/K42/>.

- [20] Steve R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 238–247, Atlanta, GA, USA, June 1986.
- [21] Insup Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, University of Wisconsin, Madison, 1983.
- [22] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *Proceedings of the Ottawa Linux Symposium*, 2002.
- [23] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, Las Vegas, NV, USA, October 1998.
- [24] Microsoft. *.NET Framework Developer's Guide*. Microsoft Press, 2005. Available from <http://msdn.microsoft.com/library/>.
- [25] Paul Russell. Module refcount & stuff mini-FAQ. Post to Linux-kernel mailing list, November 2002. <http://www.ussg.iu.edu/hypermail/linux/kernel/0211.2/0697.html>.
- [26] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, March 1993.
- [27] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–228, November 1996.
- [28] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Technical Conference*, pages 141–154, San Antonio, TX, USA, 2003.
- [29] Gareth Stoye, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtii. Mutatis Mutandis: Safe and predictable dynamic software updating. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Long Beach, CA, USA, January 2005.
- [30] Sun Microsystems Inc. *Solaris Live Upgrade 2.0 Guide*, October 2001. Available from <http://www.sun.com/software/solaris/liveupgrade/>.
- [31] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, December 2004.

SARC: Sequential Prefetching in Adaptive Replacement Cache

Binny S. Gill and Dharmendra S. Modha

IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

Emails: {binnyg,dmodha}@us.ibm.com

Abstract—Sequentiality of reference is an ubiquitous access pattern dating back at least to Multics. Sequential workloads lend themselves to highly accurate prediction and prefetching. In spite of the simplicity of the workload, design and analysis of a good sequential prefetching algorithm and associated cache replacement policy turns out to be surprisingly intricate. As first contribution, we uncover and remedy an anomaly (akin to famous Belady’s anomaly) that plagues sequential prefetching when integrated with caching. Typical workloads contain a mix of sequential and random streams. As second contribution, we design a self-tuning, low overhead, simple to implement, locally adaptive, novel cache management policy SARC that dynamically and adaptively partitions the cache space amongst sequential and random streams so as to reduce the read misses. As third contribution, we implemented SARC along with two popular state-of-the-art LRU variants on hardware for IBM’s flagship storage controller Shark. On Shark hardware with 8 GB cache and 16 RAID-5 arrays that is serving a workload akin to Storage Performance Council’s widely adopted SPC-1 benchmark, SARC consistently and dramatically outperforms the two LRU variants shifting the throughput-response time curve to the right and thus fundamentally increasing the capacity of the system. As anecdotal evidence, at the peak throughput, SARC has average response time of 5.18ms as compared to 33.35ms and 8.92ms for the two LRU variants.

I. INTRODUCTION

Moore’s law indicates that processor speed grows at an astounding 60% yearly rate. In contrast, disks which are electro-mechanical devices have improved their access times at a comparatively meager annual rate of about 8%. Moreover, disk capacity grows 100 times per decade, implying fewer available spindles for the same amount of storage [1]. These trends dictate that a processor must wait for increasingly larger number of cycles for a disk read/write to complete. A huge amount of performance literature has focused on hiding this I/O latency for disk bound applications.

A. Caching

Caching is a fundamental technique in hiding I/O latency and is widely used in storage controllers (IBM Shark, EMC Symmetrix, Hitachi Lightning), databases (IBM DB2, Oracle, SQL Server), file systems (NTFS, EXT3, NFS, CIFS), and operating systems (UNIX variants and Windows). SNIA (www.snia.org) defines a cache as “A high speed memory or storage device used to reduce the effective time required to read data from or write data to a lower speed memory or device.” We shall

study cache algorithms for a storage controller wherein fast, but relatively expensive, random access memory is used as a cache for slow, but relatively inexpensive, disks. A modern storage controller’s cache typically contains volatile memory used as a *read cache* and a non-volatile memory used as a *write cache*.

The effectiveness of read cache depends upon *hit ratio*, that is, the fraction of requests that are served from the cache without necessitating a disk trip (*miss*). We shall focus on improving the performance of the read cache that is increasing the hit ratio or equivalently minimizing the miss ratio. Typically, cache is managed in uniformly sized units called *pages*.

B. Demand Paging

Demand paging is a classical assumption used to study and design cache algorithms [2] wherein a page is brought in from the slower memory to the cache only on a miss. Demand paging precludes speculatively prefetching pages. Under demand paging, the only question of interest is: When the cache is full, and a new page must be inserted in the cache, which page should be replaced? The best, offline cache replacement policy is Belady’s MIN that replaces the page whose next access is farthest in the future [3]. In practice, variants of LRU that replaces the least recently used page [4], [5], [6] are often used. For a recent detailed survey of cache replacement policies, see [7], [8].

C. Prefetching

A deeper dent can be made in I/O latency by speculatively prefetching or prestaging pages even before they are requested [9].

To prefetch, a prediction of likely future data accesses based on past accesses is needed. The accuracy of prediction plays an important role in reducing cache pollution and in increasing the utility of prefetching. The accuracy is generally dependent upon the amount of history that can be maintained and mined on-line, and on the stationarity of the access patterns.

A number of papers have focused on predictive approaches to prefetching, for example, [10] used relationship graph models, [11] used a prediction scheme based on classical information-theoretic Lempel-Ziv algorithm, [12] used a scheme based on associative memory, and [13] used a scheme based on partitioned

context modeling. Commercial systems have rarely used very sophisticated prediction schemes. There are several reasons for this gap between research and practice. To be effective, sophisticated prediction schemes need extensive history of page accesses which is cumbersome and expensive to maintain for real-life systems. For example, a high-end storage controller may serve many tens of thousands of I/Os per second. Recording and mining such data online and in real-time is a non-trivial challenge. Furthermore, to be effective a prefetch must complete before the predicted request. This requires sufficient prior notice. However, long-term predictive accuracy is generally very low to begin with and becomes worse with interleaving of a large number of different workloads. A further degradation in predictive accuracy happens if the workloads do not exhibit stationarity which is the founding axiom behind many predictive approaches. Finally, for a disk subsystem operating near its peak capacity, average response time increases drastically with the increasing number of disk fetches. Thus, low accuracy predictive prefetching, which results in an increased number of disk fetches, can in fact worsen the performance.

In a well-known paper, [14] proposed an approach to significantly increase the predictive accuracy of prefetching by letting applications disclose their knowledge of future accesses to enable informed caching and prefetching. Building on [14], [15] utilized idle processor cycles while an application is stalled on I/O to speculatively pre-execute application's code to garner information about its future read accesses.

D. Sequential Prefetching

Sequentiality is a characteristic of workloads which reference consecutively numbered pages in ascending order without gaps. Sequential file accesses have been known at least since Multics [16]. Sequentiality naturally arises in video-on-demand, database scans, copy, backup, and recovery that may read a large number of files sequentially. Evidence of sequentiality abounds in database workloads, for example, [17], [18], [19], [20], [21]. The world of database and storage systems performance is largely dominated by benchmarks. The Transaction Processing Performance Council (TPC) benchmarks TPC-D [21], [22] and TPC-H exhibit a significant amount of sequentiality. Similarly, more recent Storage Performance Council (SPC)'s first benchmark SPC-1 is designed to be a mix of random and sequential workloads [23], [24]. The importance of sequential access patterns is further underscored by the fact that forthcoming SPC-2 benchmark will focus entirely on many concurrent sequential clients [25].

In contrast to sophisticated forecasting methods, detecting sequentiality is easy, requiring very little history

information, and can attain nearly 100% predictive accuracy. An important trend is that sequential bandwidth of the disk has been increasing at a respectable annual rate of 40% while seek time have improving only at a meager annual rate of 8%. This implies that the additional cost of read-ahead on a seek is becoming progressively smaller. For these reasons, all UNIX variants [26], most modern day file systems [27], [28], databases such as DB2 [29] and Oracle [30], and high-end storage controllers such as IBM Shark [31], EMC Symmetrix all employ sequential detection and prefetching.

E. Our Contributions

We make several contributions towards design and implementation of a self-tuning, low overhead, high performance cache replacement algorithm for a real-life system that deploys sequential prefetching. For a seemingly much studied, well understood, and "simple" technique, design and analysis of a good sequential prefetching algorithm and associated cache replacement policy turns out to be surprisingly tricky. We summarize our main novel contributions, and outline the paper:

- 1) (Section II) For a class of state-of-the-art sequential prefetching schemes that use LRU, we point out an anomaly akin to Belady's anomaly that results in more misses when larger cache is allocated. We propose a simple technique to eliminate the anomaly.
- 2) (Section III) It is common to separate sequential data and random data into two LRU lists. We develop elegant analytical and empirical models for dynamically and adaptively trading cache space between the two lists with the goal of maximizing the overall hit ratio and, consequently, minimizing the average response time. Towards this end, we synthesize a new algorithm, namely, Sequential Prefetching in Adaptive Replacement Cache (SARC), that is self-tuning, low overhead, and simple to implement. SARC improves performance for a wide range of workloads that may have a varying mix of sequential and random data streams and may possess varying temporal locality of the random data streams.
- 3) (Section IV) Shark is IBM's flagship high-end storage controller [31]. We implemented SARC and two commonly used LRU variants on Shark (Model 800) hardware. On a widely adopted SPC-1 storage benchmark, SARC convincingly outperforms the two LRU variants. As anecdotal evidence, at the same throughput, SARC has average response time of 5.18ms as compared to 33.35ms and 8.92ms for the two LRU variants.¹

II. SEQUENTIAL PREFETCHING

The goal of sequential read-ahead is to keep the cache pre-loaded and ready with data for upcoming I/O operations, thus preventing potential misses. We outline a state-of-the-art sequential prefetching scheme whose variants are used in many commercial systems, for example, DB2 [29], Oracle [30], and Shark [32]. We also point out an anomaly akin to Belady's anomaly [3] that plagues such sequential prefetching schemes when used in conjunction with LRU-based caching. We also offer a simple and elegant remedy.

We manage the lists in the cache in terms of *tracks*, where a track is a set of up to eight 4K *pages*.

Before a sequential prefetching policy can be deployed, sequential access must be detected *somehow*. In many mainframe type applications, the client often indicates sequential accesses. In the absence of such hints, a necessary first step is to effectively detect sequences. Next, we discuss one such detection scheme; if desired, other schemes can be readily substituted without changing the essence of our analysis.

A. Sequential Detection

The goal of sequential detection is to automatically uncover a sequential access pattern. Such a pattern is not obvious to discover because of possible interleaving between various sequential streams and pauses between consecutive requests by a single stream, namely, the *think time* between requests.

The basic idea is to maintain a sequential detect counter with every track, and to use a parameter `seqThreshold` that can be set differently depending upon whether aggressive or conservative sequential detection is desired. The counter is updated as follows. On a hit or miss to track n whose counter is uninitialized, if track $n - 1$ is present in the cache, then set the counter of track n to minimum of `seqThreshold` or one plus the counter value for track $n - 1$. If track $n - 1$ is not present in the cache, set the counter for track n to 1. Once initialized, the counter value for a track is not changed unless it reenters the cache after an eviction. When the counter equals `seqThreshold`, the track is termed as a *sequential track*. If a track gets designated as a sequential track on a miss, then we say that a *sequential miss* has occurred.

B. Synchronous and Asynchronous Prefetching

The simplest sequential read-ahead strategy is *synchronous prefetching* which on a sequential miss on track n simply brings tracks n through $n + m$ into the cache, where m is known as the degree of sequential read-ahead and is a tunable parameter [33]. As mentioned in the introduction, the additional cost of read-ahead on a seek is becoming progressively smaller. The

number of sequential misses decrease with increasing m , if (i) all the read-ahead tracks are accessed and (ii) not discarded before they are accessed. Consider the well known OBL (one block look-ahead) scheme [20] that uses $m = 1$. This scheme reduces the number of sequential misses by $1/2$. Generalizing, with m track look-ahead, number of sequential misses decrease by $1/(m + 1)$. To eliminate misses purely by using synchronous prefetching, m needs to become infinite. This is impractical, since prefetching too far in advance, will cause cache pollution, and will ultimately degrade performance. Also, depending upon the amount of cache space available to sequential data, not all tracks can be used before they are evicted. Hence, by simply increasing m , it is not always possible to drive the number of sequential misses to zero. The behavior of sequential misses for synchronous prefetching is illustrated in the left-hand panel of Figure 1.

To effectively eradicate misses, asynchronous prefetches can be used [29], [32]. An *asynchronous prefetch* is carried out in the absence of a sequential miss, typically, on a hit. The basic idea is to read-ahead a first group of tracks synchronously, and after that when a preset fraction of the prefetched group of tracks is accessed, *asynchronously* (meaning in the absence of a sequential miss) read-ahead next group of tracks, and so forth and so on. Typically, asynchronous prefetching is done on an *asynchronous trigger*, namely, a special track in a prefetched group of tracks. When the asynchronous trigger track is accessed, the next group of tracks is asynchronously read-ahead and a new asynchronous trigger is set. The intent is to exploit sequential structure to continuously stage tracks ahead of their access without incurring a single additional miss other than the initial sequential miss. A good analogy is that of an on-going relay race where each group of tracks passes the baton on to the next group of tracks and so on.

To summarize, in state-of-the-art sequential prefetching, synchronous prefetching is used initially when the sequence is first detected. After this bootstrapping stage, asynchronous prefetching can sustain itself as long as all the tracks within the current prefetched group are accessed before they are evicted. As a corollary, the asynchronous trigger track will also be accessed, and in turn, will prefetch the next group of tracks amongst which will also be the next asynchronous trigger.

C. Combining Caching and Prefetching for Sequential Data

So far, we have discussed two crucial aspects of sequential prefetching: (i) What to prefetch? and (ii) When to prefetch? We now turn our attention to the next issue, namely, management of prefetched data in the

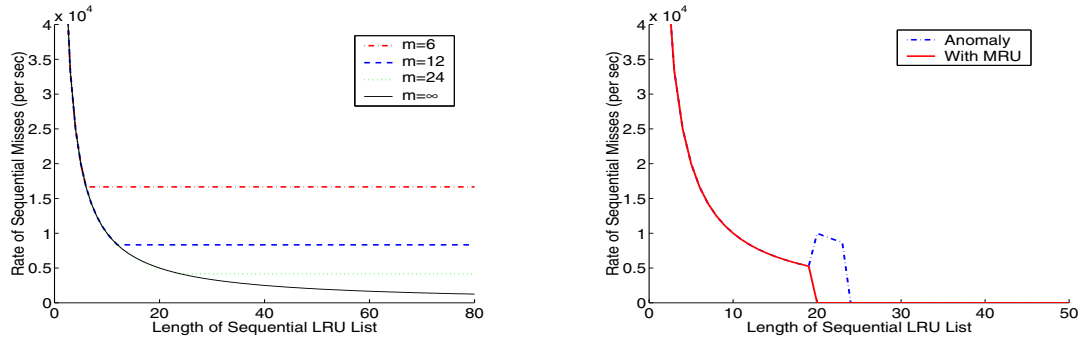


Fig. 1. Both of the above graphs were obtained via a simulation for a single sequential stream. The left-hand panel depicts behavior of sequential misses for synchronous prefetching. The lower most hyperbolic curve corresponds to an idealized situation with an infinite degree of read-ahead. It can be seen that for a fixed, finite degree of read-ahead m , sequential misses initially follow the hyperbolic curve, and, then become constant. Moreover, higher the degree of read-ahead, the smaller the minimum constant attained. The right-hand panel depicts behavior of sequential misses for synchronous+asynchronous prefetching with and without an anomaly. The anomalous curve initially follows the hyperbolic curve, but has a bump before going to zero.

cache. Given a fixed amount of cache, prefetched data cannot be kept forever and must eventually be replaced. Prefetching and caching are intertwined, and cannot be studied in isolation.

In practice, the most widely used online policy for cache management is LRU that maintains a doubly-linked list of tracks according to recency of access. In the context of sequential prefetching, when tracks are prefetched or accessed, they are placed at the MRU (most recently used) end of the list. And, for cache replacement, tracks are evicted from the LRU end of the list.

We now describe an interesting situation that arises when the above described synchronous+asynchronous prefetching strategy is used along with the LRU-based caching. Suppose that the asynchronous trigger track in a currently active group of prefetched tracks is accessed. This will cause an asynchronous prefetch of the next group of tracks. In an LRU-based cache, these newly fetched group of tracks along with the asynchronous trigger track will be placed at the MRU end of the list. The unaccessed tracks within the current prefetch group remain where they were in the LRU list, and, hence, in some cases, could be near the LRU end of the list. However, observe that these unaccessed tracks within the current prefetch group will be accessed before the tracks in the newly prefetched group. Hence, depending upon the amount of cache space available for sequential data, it can happen that some of these unaccessed tracks may be evicted from the cache before they are accessed resulting in a sequential miss. Furthermore, this may happen repeatedly, thus defeating the purpose of employing asynchronous prefetching. This observation is related to “Do No Harm” rule of [34] in the context of offline policies for integrated caching and prefetching.

In a purely demand-paging context, LRU is a *stack*

algorithm [2]. Quite surprisingly, when LRU-based caching is used along with the above prefetching strategy, the resulting algorithm violates the stack property. As a result, when the amount of cache space given to sequentially prefetched data increases, sequential misses do not necessarily decrease. This anomaly is illustrated in the right-hand panel of Figure 1. As will be seen in the sequel, a stack property is a crucial ingredient in our algorithm.

At the cost of increasing sequential misses, both of the above problems can be hidden if (i) only synchronous prefetching is used or (ii) both synchronous+asynchronous prefetching are used and the asynchronous trigger is always set to be the last track in a prefetched group. The first approach amounts to a relapse in which we forego all potential benefits of asynchronous prefetching. The second approach is attractive in principle; however, if the track being prefetched is accessed before it is in the cache, then the resulting sequential miss will not be avoided. To avoid this sequential miss, in real life, the asynchronous trigger track is set sufficiently before the last prefetched track so that the next prefetched group arrives in cache before it is actually accessed.

Unlike the above two approaches, we are interested in fixing the anomalous behavior without incurring additional sequential misses. To this end, we now propose a simple to implement and elegant algorithmic enhancement. As mentioned above, in an LRU-based cache, the newly prefetched group of tracks along with the asynchronous trigger track in the current group of tracks are placed at the MRU end of the list. We propose, in addition, to also move all unaccessed tracks in the current group of tracks to the MRU end of the list. As can be seen in the right-hand panel of Figure 1, this enhancement retains all benefits of asynchronous

prefetching while ridding it of its anomalous behavior.

III. SARC: SEQUENTIAL PREFETCHING IN ADAPTIVE REPLACEMENT CACHE

So far, we have focused primarily on designing an effective sequential prefetching strategy along with an LRU-based caching policy for housing sequential data. Typical workloads contain a mix of sequential and random streams. We now turn our attention to designing an integrated cache replacement policy that manages the cache space amongst both of these classes of workloads so as to minimize the overall miss rate.

A. Prior Work

A number of approaches have been proposed for integrated caching of sequential and random data. Almost all of them employ an LRU variant. One simple approach [20] that we call LRU-Top is to maintain a single LRU list for both sequential and random data, and whenever tracks are prefetched or accessed, they are placed at the MRU end of the list. The tracks are evicted from the LRU end of the list. Another approach [35], [36] that we call LRU-Bottom is to maintain a single LRU list for both sequential and random data; however, while random data is inserted at the MRU end of the list, sequential data is inserted near the LRU end. For another interesting LRU variant, see [37]. [1] suggested holding sequential data for a fixed amount of time, while [29] suggested giving sequential data a fixed, predetermined portion of the cache space. [34] studied offline, optimal policies for integrated caching and prefetching. In a recent work, [38] focused on general demand pre-paging and noted that the amount of cache space devoted to prefetched data is “critical, and its ideal value depends not only on the predictor and the degree, but also on the main memory size, the application, and the reference behavior of the process.” In other words, no strategy that is independent of the workload characteristics is likely to be universally useful.

In the context of demand paging, in addition to LRU, a number of cache replacement policies have been studied, see, for example, LFU, FBR, LRU-2, 2Q, MQ, LRFU, and ARC. For a detailed overview of these algorithms, see [7], [8]. Our context is different than that of these algorithms, since we are interested in integrated policies for caching and prefetching. Previously, [14] have considered adaptively balancing cache amongst three partitions: LRU cache, hinted cache, and prefetch cache. It is not clear how to efficiently extend the algorithm in [14] in presence of potentially a very large number of sequential streams.

Our algorithm, namely, Sequential Prefetching in Adaptive Replacement Cache (SARC), is closely related to—but distinct from—Adaptive Replacement Cache

(ARC). In particular, the idea of two adaptive lists in SARC is inspired by ARC. There are several differences between the two algorithms: (i) ARC is applicable only in a demand paging scenario, whereas SARC combines caching along with sequential prefetching. (ii) While ARC maintains a history of recently evicted pages, SARC does not need history and is also simpler to implement.

B. Our Approach

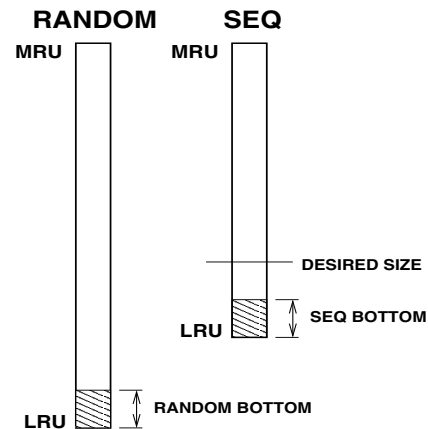


Fig. 2. SARC separates sequential and random data into two lists, and maintains a desired size parameter for the sequential list. The desired size is continually adapted in response to a dynamic, changing workload. Specifically, if the bottom portion of SEQ list is found to be more valuable than the bottom portion of RANDOM list, then the desired size is increased; otherwise, the desired size is decreased.

We shall develop an adaptive, self-tuning, low overhead algorithm that dynamically partitions the amount of cache space amongst sequential and random data so as to minimize the overall miss rate. Given the radically different nature of sequentially prefetched pages and the random data, it is natural to separate these two types of data. As shown in Figure 2, we will manage each class in separate LRU lists: RANDOM and SEQ. One of our goals is to avoid *thrashing* [34] that happens when more precious demand-paged random pages are replaced with less precious prefetched pages (cache pollution) or when prefetched pages are replaced too early before they are used.

A key idea in our algorithm is to maintain a desired size (see Figure 2) for SEQ list. The tracks are evicted from the LRU end of SEQ, if its size (in pages) is larger than the desired size; otherwise, the tracks are evicted from the LRU end of RANDOM. The desired size is continuously adapted. We now explain the intuition behind this adaptation.

Assuming that both the lists satisfy the LRU stack property (see Section II-C), the optimum partition is one that equalizes the marginal utility of allocating additional cache space to each list. We design a *locally*

adaptive algorithm that starts from any given cache partitioning and gradually and dynamically tweaks it to gear it towards the optimum. As an important step towards this goal, we first derive an elegant analytical model for computing the marginal utility of allocating additional cache space to sequentially prefetched data. In other words, how does the number of sequential misses experienced by **SEQ** change as the size of the list changes. Similarly, as a second step, we empirically estimate the marginal utility of allocating additional cache space to random data. Finally, as the third step, if during a certain time interval the marginal utility of **SEQ** list is higher than that of **RANDOM** list, then the desired size is increased; otherwise, the desired size is decreased.

C. Single Sequential Stream

For simplicity, assume that there is only one request to each track. Multiple consecutive requests do not change the final algorithm in any way.

Every track in cache has a time stamp that is updated with the current time whenever the track is placed at the **MRU** position of either list. Let T denote the *temporal length*, that is, the time difference between the **MRU** and **LRU** time stamps of **SEQ**.

Let s_1 and s_1^a denote the rates of sequential misses of one stream when synchronous and synchronous+asynchronous prefetching, respectively, are employed.

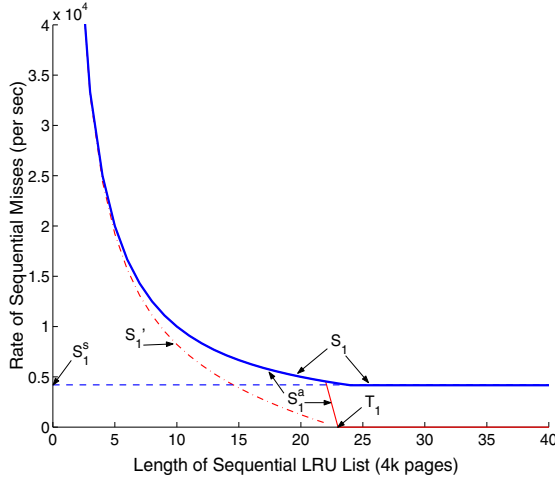


Fig. 3. Via simulation for a single sequential stream, we depict behavior of s_1 (rate of sequential misses for synchronous prefetching), s_1^a (rate of sequential misses for synchronous+asynchronous prefetching), and s_1' (an approximation to s_1^a). The hyperbolic curve s_1 flattens out at point (T_1, s_1^s) .

Figure 3 displays the behavior of s_1 and s_1^a as the temporal length varies. As discussed in Section II-B, under synchronous prefetching, sequential misses or its

rate is inversely proportional to the degree of read-ahead. If, however, the read-ahead are discarded before they are accessed, the *effective degree of read-ahead* decreases. Whenever the effective degree of read-ahead is less than the actual degree, it is proportional to the temporal length of the list. Hence, we have that

$$s_1 = \begin{cases} (\text{constant})/T & 0 \leq T \leq T_1 \\ s_1^s & T_1 < T, \end{cases} \quad (1)$$

where $s_1^s > 0$ is the smallest attainable rate of sequential misses (which is inversely proportional to the actual degree of read-ahead) and T_1 is the smallest temporal length that attains s_1^s . As previously discussed in Section II-B and depicted in Figure 3, s_1^a drops to zero when s_1 flattens out:

$$s_1^a = \begin{cases} s_1 & 0 \leq T \leq T_1 \\ 0 & T_1 < T. \end{cases} \quad (2)$$

For later use, we define the curve s_1' as:

$$s_1' = \begin{cases} s_1 - T(s_1^s)/(T_1) & 0 \leq T \leq T_1 \\ 0 & T_1 < T. \end{cases} \quad (3)$$

Observe that the curve s_1' distributes the discontinuity of s_1^a throughout the interval $[0, T_1]$.

D. Marginal Utility of SEQ

When we have multiple streams, their think times will in general be different. Also, in general, each client will have different number of accesses to each track before moving onto the next track. Fortunately, these differences do not enter our analysis below.

Let us suppose that there are ℓ streams. The parameter ℓ will not appear anywhere in our algorithm. Let s_i^a , $1 \leq i \leq \ell$, denote the rate of sequential misses of stream i for synchronous+asynchronous prefetching. Observe that these individual numbers are generally not easily observable. However, their sum

$$s = \sum_{i=1}^{\ell} s_i^a \quad (4)$$

is simple to observe in a real system.

Let L denote the length of the list **SEQ** in number of 4K pages. To compute the marginal utility of **SEQ**, we examine how the overall rate of sequential misses changes, namely, Δs , in response to a small change ΔL in L . By the stack property of **SEQ**, as L increases (respectively, decreases) s decreases (respectively, increases). Hence, $\Delta s / \Delta L$ is always negative. The marginal utility is measured by the magnitude of this quantity. We shall lower bound this negative

quantity, and, in turn, upper bound the marginal utility.

$$\begin{aligned}
\frac{\Delta s}{\Delta L} &= \frac{\Delta T}{\Delta L} \frac{\Delta s}{\Delta T} \\
&\stackrel{(a)}{=} \frac{\Delta T}{\Delta L} \sum_{i=1}^{\ell} \frac{\Delta s_i^a}{\Delta T} \\
&\stackrel{(b)}{\geq} \frac{\Delta T}{\Delta L} \sum_{i=1}^{\ell} \frac{\Delta s_i'}{\Delta T} \\
&\stackrel{(c)}{=} \frac{\Delta T}{\Delta L} \sum_{i=1}^{\ell} I(T < T_i) \cdot \left(\frac{\Delta s_i}{\Delta T} - \frac{s_i^s}{T_i} \right) \\
&\stackrel{(d)}{=} \frac{\Delta T}{\Delta L} \sum_{i=1}^{\ell} I(T < T_i) \cdot \left(-\frac{s_i}{T} - \frac{s_i}{T} \left(\frac{T}{T_i} \right)^2 \right) \\
&\stackrel{(e)}{>} \frac{\Delta T}{\Delta L} \sum_{i=1}^{\ell} \left(-2 \frac{s_i^a}{T} \right) \\
&\stackrel{(f)}{=} -2 \frac{\Delta T}{\Delta L} \frac{s}{T} \\
&= -2 \frac{L \Delta T}{T \Delta L} \frac{s}{L} \\
&\stackrel{(g)}{=} -2 \frac{s}{L}, \tag{5}
\end{aligned}$$

where (a) follows from (4); (b) follows since, by definitions in (2) and (3), s_i' that is steeper than s_i^a throughout the continuous region $[0, T_i]$; (c) follows from (3) and I is the indicator function that takes value 1 or 0 depending upon whether $T < T_i$ or not; (d) follows from (1); (e) follows since $T < T_i$ and the indicator function $I(T < T_i)$ can be removed since when I is zero s_i^a is also zero; (f) follows from (4); and (g) follows since, in practice, there is linear relationship between L and T . In other words, as the length of the list increases (respectively, decreases) the time difference between the MRU and LRU time stamps of the list increases (respectively, decreases) proportionately for any given workload.

Let T_i , $1 \leq i \leq \ell$, denote the times at which various streams attain zero misses (see Figure 3). Mathematically, the above chain of inequalities is valid at all points in $[0, \infty)$ except at T_i , $1 \leq i \leq \ell$, since at T_i , $\Delta s_i^a / \Delta T$ is not defined. However, our choice of the approximating curve s_i' is such that it cleverly distributes the magnitude of the drop in s_i^a at T_i evenly throughout the interval $[0, T_i]$, and, hence, in practice, the inequalities are applicable. This approximation smoothes out the changes in Δs in relation to ΔL at the discontinuities and paves the way for designing a locally adaptive algorithm such as ours.

We now have the following bounds

$$-\frac{s}{L} \geq \frac{\Delta s}{\Delta L} \geq -2 \frac{s}{L},$$

where the upper bound follows from (1) and (2) and

the lower bound follows from (5). In other words, the marginal utility lies somewhere between s/L and $2s/L$, but closer to latter in practice. In this paper, we have chosen $2s/L$ to approximate the marginal utility of SEQ.

E. Marginal Utility of RANDOM

The RANDOM list contains essentially only demand-paged, non-sequential data. For demand paged data, LRU is a stack algorithm. In other words, increasing (respectively, decreasing) the length of the list leads to smaller (respectively, larger) number of misses.

Let r denote the rate of cache misses in RANDOM. To compute the marginal utility of RANDOM, we examine r changes, namely, Δr , changes in response to a small change ΔL in L . By the stack property, as L increases (respectively, decreases) r decreases (respectively, increases). Hence, $\Delta r / \Delta L$ is always negative. The marginal utility is measured by the magnitude of this quantity. Unlike SEQ list where an analytical model was necessary, for the RANDOM list the quantity $\Delta r / \Delta L$ can be estimated directly from real-time cache introspection.

We will monitor ΔL cache space at the bottom of RANDOM list (see Figure 2), and observe rate of hits Δr in this region. A bottom hit is an indication that a miss has been saved due to the cache space at the bottom of RANDOM. In other words, if cache space ΔL at the bottom was not allocated to RANDOM, then the rate of misses r would increase by Δr . We assume that $\Delta r / \Delta L$ is a constant in a small region (generally much smaller than even ΔL) at the bottom of the list where our locally adaptive algorithm is active. Hence, as a corollary, if a small amount of cache space were added to RANDOM, then the misses would decrease in proportion to $\Delta r / \Delta L$.

However, allocating more cache space to RANDOM will take away equal amount from SEQ, and, hence, needs to be carefully weighed. The optimum is attained when marginal utilities of both the lists are equalized.

F. Equalizing Marginal Utilities

Figure 2 depicts the structure of our algorithm. Fix the size of RANDOM bottom ΔL to a small constant fraction of the cache size. The essence of the algorithm is to compare the absolute values of

$$\frac{\Delta s}{\Delta L} \left(\approx \frac{2 \cdot s}{L} \right) \text{ and } \frac{\Delta r}{\Delta L}.$$

If $(2s)/L$ is larger than the magnitude of $\Delta r / \Delta L$, then it is more advantageous to transfer cache space from the bottom of RANDOM to the bottom of SEQ and hence we increase the desired size of SEQ; otherwise, we decrease the desired size of SEQ.

So far, the time interval for sampling the rates Δr and s has not been fixed. The smaller the time interval the more adaptive the algorithm, while larger the time interval the smoother and slower the adaptation. Our algorithm implicitly selects a time interval based on cache hits. Thus, the rate of adaptation is derived from and adapts to the workload itself. Specifically, we select the time interval to be the time difference between two successive hits in the bottom of the **RANDOM** list. In this case, Δr is just a constant 1, and we measure s during the same interval. Thus, we now need to simply compare

$$\frac{2 \cdot s}{L} \text{ and } \frac{1}{\Delta L}, \text{ or, equivalently, } \frac{2 \cdot s \cdot \Delta L}{L} \text{ and } 1. \quad (6)$$

G. SARC

We now weave together the above analysis and insights into a concrete, adaptive (self-tuning) algorithm that dynamically balances the cache space allocated to **RANDOM** and **SEQ** data under different conditions such as different number of sequential/random streams, different data layout strategies, different think times of sequential/random streams, different cache-sensitivities/footprints of random streams, and different I/O intensities. The complete algorithm is displayed in Figure 4. **SARC** is extremely simple to implement and has virtually no computational/space overhead over an LRU-based implementation.

In our experiments, we have set ΔL to be 2% of the cache space. Any other small fraction would work as well.

In the algorithm, we will need to determine if a hit in one of the lists was actually in its bottom (see lines 6 and 12). Of course, one could maintain separate fixed sized bottom lists for both the lists. However, with an eye towards simplification and computational efficiency, we now describe a time-based approximation to achieve nearly the same effect. We now describe how to determine if the a hit in **SEQ** lies in its bottom ΔL . Let T_{MRU} and T_{LRU} denote the time stamp of the MRU and LRU tracks, respectively, in **SEQ**. Let L denote the size of **SEQ** in pages. Let T_{HIT} denote the time stamp of the hit track. Now, if

$$(T_{HIT} - T_{LRU}) \leq \frac{\Delta L}{L}(T_{MRU} - T_{LRU}),$$

then we say that a bottom hit has occurred. The same calculation can also be used for determining the bottom hits in **RANDOM**.

The algorithm uses the following constants: m (lines 18 and 32) denotes the degree of synchronous and asynchronous read-ahead; g (lines 18 and 32) denotes the number of disks in a RAID group; `triggerOffset` (line 49) is the offset from the end of a prefetched group

of tracks and is used to demarcate the asynchronous trigger; `LargeRatio` (line 13) is a threshold, say, 20, to indicate that the `ratio` has become much larger than 1; and `FreeQThreshold` (line 52) denotes the desired size of the `FreeQ`. **Shark** uses RAID arrays (either RAID-5 or RAID-10) at the back-end. Our machine (see Section IV-B) was configured with RAID-5 (6 data disks + parity disk + spare disk) meaning that $g = 6$. RAID allows for parallel reads since logical data is striped across the physical data disks. To leverage this, setting m as a multiple of g is meaningful. We set $m = 24$. Finally, we chose `triggerOffset` to be 3.

The algorithm is self-explanatory. We now briefly explain important portions of the algorithm in Figure 4 in a line-by-line fashion.

Lines 1-3 are used during the initialization phase only. The counter `seqMiss` tracks the number of sequential misses between two consecutive bottom hits in **RANDOM**, and is initialized to zero. The variable `desiredSeqListSize` is the desired size of **SEQ**, and is initially set to zero meaning adaptation is shut off. The adaptation will start only after **SEQ** is populated (see lines 69-73). The variable `adapt` determines the instantaneous magnitude and direction of the adaptation to `desiredSeqListSize`.

Lines 4-50 describe the cache management policy. The quantity `ratio` in line 4 is derived from (6). Lines 5-10 deal with the case when a track in **RANDOM** is hit. If the hit is in the bottom portion of the **RANDOM** list (line 6), then `seqMiss` is reset to zero (line 7) since we are interested in number of sequential misses between two successive hits in the bottom of **RANDOM**. Line 8, sets the variable

$$\text{adapt} = \frac{2 \cdot \text{seqMiss} \cdot \Delta L}{L} - 1.$$

Note that in the steady state the rate of new tracks being added to any cache is the same as the rate of old tracks being demoted from the cache. This rate is an upper bound on the rate at which the size of either **SEQ** or **RANDOM** can change while keeping the total number of tracks in the cache constant. Since `adapt` is used to change the `desiredSeqListSize`, it is therefore not allowed to exceed 1 or be less than -1. Observe that by the test prescribed in (6), if `adapt` is greater than zero, then we would like to increase `desiredSeqListSize`, else we would like to decrease it. This increase or decrease is executed in line 70. Also, observe that when the inequality between the marginal utilities of **SEQ** and **RANDOM** is larger, the magnitude of `adapt` is larger, and, hence, a faster rate of adaptation is adopted, whereas when the two marginal utilities are nearly equal, `adapt` will be close to zero, and a slower rate of adaptation is adopted. Finally, observe that line 70 (which carries out the actual adaptation) is executed

INITIALIZATION: Set the adaptation variables to 0.

- 1: Set seqMiss to 0
 - 2: Set adapt to 0
 - 3: Set desiredSeqListSize to 0
-

CACHE MANAGEMENT POLICY:

Track x is requested:

- 4: Set ratio = $(2 \cdot \text{seqMiss} \cdot \Delta L) / \text{seqListSize}$
 - 5: case i: $x \in \text{RANDOM}$ (HIT)
 - 6: if $x \in \text{RANDOM BOTTOM}$ then
 - 7: Reset seqMiss = 0
 - 8: Set adapt = $\max(-1, \min(\text{ratio} - 1, 1))$
 - 9: endif
 - 10: Mru(x , RANDOM)
 - 11: case ii: $x \in \text{SEQ}$ (HIT)
 - 12: if $x \in \text{SEQ BOTTOM}$ then
 - 13: if (ratio > LargeRatio) then
 - 14: Set adapt = 1
 - 15: endif
 - 16: endif
 - 17: if x is AsyncTrigger then
 - 18: ReadAndMru($[x + 1, x + m - x \% g]$, SEQ)
 - 19: endif
 - 20: Mru(x , SEQ)
 - 21: if track $(x - 1) \in (\text{SEQ} \cup \text{RANDOM})$ then
 - 22: if (seqCounter($x - 1$) == 0) then
 - 23: Set seqCounter(x) = $\max(\text{seqThreshold}, \text{seqCounter}(x - 1) + 1)$
 - 24: endif
 - 25: else
 - 26: Set seqCounter(x) = 1
 - 27: endif
 - 28: case iii: $x \notin (\text{SEQ} \cup \text{RANDOM})$ (MISS)
 - 29: if $(x - 1) \in (\text{SEQ} \cup \text{RANDOM})$ then
 - 30: if seqCounter($x - 1$) == seqThreshold then
 - 31: seqMiss++
 - 32: ReadAndMru($[x, x + m - x \% g]$, SEQ)
 - 33: Set seqCounter(x) = seqThreshold
 - 34: else
 - 35: ReadAndMru($[x, x]$, RANDOM)
 - 36: Set seqCounter(x) = seqCounter($x - 1$) + 1
 - 37: endif
 - 38: else
 - 39: Set seqCounter(x) = 1
 - 40: endif
-

CACHE MANAGEMENT POLICY (CONTINUED):

- ReadAndMru([start, end], listType)
- 41: foreach track t in [start, end]; do
 - 42: if $t \notin (\text{SEQ} \cup \text{RANDOM})$ then
 - 43: grab a free track from FreeQ
 - 44: read track t from disk
 - 45: endif
 - 46: Mru(t , listType)
 - 47: done
 - 48: if (listType == SEQ)
 - 49: Set AsyncTrigger as (end - triggerOffset)
 - 50: endif
-

FREE QUEUE MANAGEMENT:

- FreeQThread()
- 51: while (true) do
 - 52: if length(FreeQ) < FreeQThreshold then
 - 53: if (seqListSize < ΔL
 - 54: or randomListSize < ΔL) then
 - 55: if (lru track of SEQ is older than
 - 56: lru track of RANDOM) then
 - 57: EvictLruTrackAndAdapt(SEQ)
 - 58: else
 - 59: EvictLruTrackAndAdapt(RANDOM)
 - 60: endif
 - 61: else
 - 62: if (seqListSize > desiredSeqListSize) then
 - 63: EvictLruTrackAndAdapt(SEQ)
 - 64: else
 - 65: EvictLruTrackAndAdapt(RANDOM)
 - 66: endif
 - 67: endif
 - 68: endif
 - 69: evict lru track in listType and add it to FreeQ
 - 70: if (desiredSeqListSize > 0) then
 - 71: Set desiredSeqListSize += adapt / 2
 - 72: else
 - 73: Set desiredSeqListSize = seqListSize
 - 74: endif
 - 75: endif
-

Fig. 4. Algorithm for Sequential Prefetching Adaptive Replacement Cache. This algorithm is completely self-contained, and can directly be used as a basis for an implementation. The algorithm starts from an empty cache.

only when a track is actually evicted from one of the lists. In a steady state, tracks are evicted from the cache at the rate of cache misses. Hence, a larger (respectively, a smaller) rate of misses will effect a faster (respectively, a slower) rate of adaptation. Hence, **SARC** adapts not only the sizes of the two lists, but also the rate at which the sizes are adapted.

Lines 11-27 deal with the case when a track in **SEQ** is hit. If the hit is in the bottom portion of the **SEQ** list (line 12) and ratio has become large (line 13), in other words, no hit has been observed in the bottom of the **RANDOM** list while comparatively large number of sequential misses have been seen on the **SEQ** list, then set adapt to 1 (line 14) meaning that increase desiredSeqListSize at the fastest rate possible. Now, if the hit track is an asynchronous trigger track (line 17), then asynchronously read-ahead the next sequential group of tracks (line 18). Lines 21-27 describe how the sequential detection mechanism in Section II-A is implemented.

Lines 28-40 deal with a cache miss. For a sequential miss (lines 29-31), synchronously read-ahead a sequential group of tracks (line 32). The remaining lines deal with sequential detection mechanism in Section II-A.

Lines 41-50 (i) read the missing tracks from a given range of tracks; (ii) places all tracks in the given range at the **MRU** position; and (iii) set the asynchronous trigger.

Lines 51-73 implement the cache replacement policy and carry out the adaptation. As is typical in multi-threaded systems, we assume that these lines run on a separate thread (line 51). If the size of the free queue drops below some predetermined threshold (line 52), then tracks are evicted from **SEQ** if it exceeds desiredSeqListSize and tracks are evicted from **RANDOM** otherwise. In either case, the evicted tracks are placed on the free queue. Observe that **SARC** becomes active (lines 60-64) only if the sizes of both the **SEQ** and the **RANDOM** list exceed ΔL . Otherwise, a simple **LRU** eviction (lines 54-58) is done. Whenever the utility of one of the two lists becomes so small when compared to the utility of the other list that its size eventually drops below ΔL , we do not want to waste even ΔL amount of cache, and revert back to **LRU**. Whenever both list sizes exceed ΔL , **SARC** takes over. Finally, lines 68-73 evict the **LRU** track from the desired list, and effect an adaption as already described above.

Our description of **SARC** is now complete.

IV. SYSTEM IMPLEMENTATION, WORKLOAD, AND RESULTS

We implemented **SARC** and two well known **LRU** variants, namely, **LRU Top** and **LRU Bottom** (see Section III-A), on IBM Shark (formally, TotalStorage Enterprise Storage Server Model 800) hardware. We

compare the performance of **SARC** to the **LRU** variants on an SPC-1 Like benchmark workload in different configurations.

A. Shark

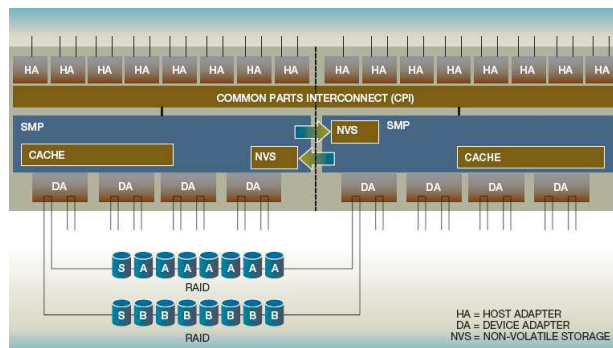


Fig. 5. A conceptual representation of IBM Shark (Enterprise Storage Server 800) [39].

Figure 5 outlines the architecture of Shark. The architecture can support upto 16 host adapters (HA) in four host adapter bays. The host adapters can have fiber channel, ESCON, or SCSI ports. Shark has two active cluster processors with symmetrical multiprocessors (SMP) for performance, reliability, and availability. Each host adapter is connected to both the SMP clusters via the Common Parts Interconnect (CPI). Either cluster is able to handle IOs from any host adapter. Both the clusters have multiple SMPs in processor drawers and have an I/O drawer which provide PCI connections for access to non-volatile, battery-backed memory (denoted as NVS in the figure) and the device adapters (denoted as DA in the figure). The processor drawer also contains up to 32GB cache per cluster. For read data, the host adapter directs the request to the appropriate cluster. For write data, the data is written to both the clusters: on one it resides in the SMP RAM and on the other cluster it resides in the NVS memory. At the back-end, Shark uses RAID arrays (or arrays) that can be configured as RAID-5 or RAID-10. For further details on Shark, please see [31], [39].

B. Our Experimental Setup

Our Shark was equipped with: 8 GB cache (per cluster), 2 GB NVS (per cluster), four 600 MHz PowerPC/RS64IV CPUs (per cluster), and 16 RAID-5 (6 + parity + spare) arrays with 72 GB, 10K rpm drives. We use only one cluster since enhancements provided by dual cluster are not necessary to study effectiveness of cache algorithms. In a single cluster mode, the behavior of our experimental software on Shark does not change other than the fact that writes now go to both the NVS memory as well as the SMP RAM in the same cluster.

	Warm-up Phase	Measurement Phase					
Time (in minutes)	120	30	30	30	30	30	30
Load (percentages)	100	100	97.5	95	80	50	10
High Load (scaled IOPS)	25000	25000	23750	22500	20000	12500	2500
Low Load (scaled IOPS)	11364	11364	10795	10227	9091	5682	1136

TABLE I. The structure of the SPC-1 Like benchmark on two different loads. For both loads, the warm-up phase is run for 120 minutes followed by 6 measurement phases of 30 minutes each. The purpose of the warm-up phase is to fill-up the cache and to bring it to a steady-state. Observe that in the measurement phase, we gradually decrease the load in proportion 100%, 97.5%, 95%, 80%, 50%, and 10%, where 100% represents the highest load. This allows studying the behavior of the storage controller under a wide range of load conditions.

The essence of our results does not change with a larger or a smaller cache size.

As a client, we use a powerful AIX host with the following configuration: 16 GB RAM, 2-way SMP with 1GHz PowerPC/Power4 CPUs. The host is connected to the Shark through two fiber channel cards which can support more than sufficient bandwidth for our all experiments.

C. SPC-1 Like Workload

SPC-1 is a synthetic, but sophisticated and fairly realistic, performance measurement workload for storage subsystems used in business critical applications. The benchmark simulates real world environments as seen by on-line, non-volatile storage in a typical server class computer system. SPC-1 measures the performance of a storage subsystem by presenting to it a set of I/O operations that are typical for business critical applications like OLTP systems, database systems and mail server applications. For extensive details on SPC-1, please see: [23], [24]. We used SPC-1 Like that is an earlier prototype implementation of SPC-1 benchmark by Bruce McNutt who was one of the chief architects of the official SPC-1 benchmark.

The SPC-1 Like workload synthesizes a community of users targeting I/Os to the storage that is logically organized in the form of three Application Storage Units (ASU). ASU-1 represents a “Data Store”, ASU-2 represents a “User Store”, and ASU-3 represents a “Log/Sequential Write”. Of the total amount of available back-end storage, 45% is assigned to ASU-1, 45% is assigned to ASU-2, and remaining 10% is assigned to ASU-3 as per SPC-1 specifications. We shall furnish more details on sizes of various ASUs in Section IV-D.

The SPC-1 Like workload is specified in units of Business Scaling Units (BSU). One BSU corresponds to a community of users who collectively generate up to 50 IOPS. The overall composition of a BSU, and, that of SPC-1 Like, is specified by the following simple matrix, where all numbers are in percentages:

	Read	Write	All
Random	29	32	61
Sequential	11	28	39
All	40	60	100

The workload is scaled by using more BSUs that in effect increases the number of users being simulated.

In this paper, due to the commercial nature of the system involved, we will not use IOPS, but rather use *scaled IOPS* which are obtained by multiplying the true IOPS by a constant (a non-integer rational number) that is not revealed in the paper.

We shall use two different load schedules for SPC-1 Like in Table I.

D. Footprint of the Workload

While modern storage controllers can make available immense amount of space, in a real-life scenario, workloads actively use only a fraction of the total available storage space known as the *footprint*. Generally speaking, for random workloads, for a given cache size, the larger the footprint, the smaller the hit ratio, and vice versa. In this paper, we will use the following two different back-end configurations in conjunction with the SPC-1 Like workload:

	ASU-1	ASU-2	ASU-3
(percentages)	45	45	10
cache-sensitive (GB)	45	45	10
cache-insensitive (GB)	1443	1443	320

E. Data Layout: Striping

RAID stripes data across its constituent disks. In addition, our AIX host permits striping data across RAID arrays. There are essentially three striping models: (i) wide striping; (ii) narrow striping; and (iii) no striping. Wide striping lays out contiguous data across all the arrays whereas narrow striping lays out data across only a subset of the arrays. For a detailed study comparing wide striping to narrow striping, please see [40]. For a very useful practical introduction to the mechanics of implementing striping, please see [41]. For random streams, striping is useful in reducing “hot

spots” (points of contention) leading to a reduction in the average response time for random seeks.

While wide striping is extremely useful for random clients, it is not friendly to sequential clients. Because striping is done at the host level and is not visible to Shark, when wide striped, each sequential access stream could appear as multiple, although slower, sequential access streams. Narrow striping is likely to have moderate number of hot spots for random clients, but is friendlier to sequential clients. In this paper, we have used narrow striping that stripes across 4 RAID arrays. Our insights, analysis, and algorithm do not change with wide striping or with no striping.

F. Results

We now compare performance of SARC to two state-of-the-art LRU variants, namely, LRU Top and LRU Bottom (see Section III-A), using the SPC-1 Like benchmark on Shark hardware that is running our experimental software implementations.

SARC minimizes the number of misses. The effect of such minimization can be studied from two perspectives, namely, that of the client and that of the storage controller. To a client, at a fixed load, the most important metric is the average response time. To study the performance seen by a client over a large spectrum of real-life operating scenarios, for the SPC-1 Like benchmark, we compare throughput versus average response time curves for all the three algorithms. To a storage controller, a crucial metric is the internal load on the RAID arrays. Within Shark, we measure the rate of tracks being staged to the cache due to read requests (both random and sequential). We will demonstrate how SARC convincingly outperforms both the LRU variants from the perspective of the client as well as the storage controller.

1) *Throughput vs. Response Time:* The two plots in the left column of Figure 6 show the throughput (in scaled IOPS) versus average response time (in ms) for all three algorithms by using the SPC-1 Like workload. Each displayed data point is an average of 27 numbers, each number being an overall response time average for read and write requests over a minute. According to SPC-1 specification, the numbers corresponding to the first three minutes of a measurement phase are discarded.

The top, left plot is obtained on a cache-sensitive configuration (see Section IV-D) for which, due to relatively high cache hit ratio, a High Load schedule (see Table I) is required to saturate the machine. The bottom, left plot is observed on a cache-insensitive configuration for which, due to relatively low cache hit ratio, a Low Load schedule is sufficient. LRU Bottom generally allocates more cache space to RANDOM than to SEQ

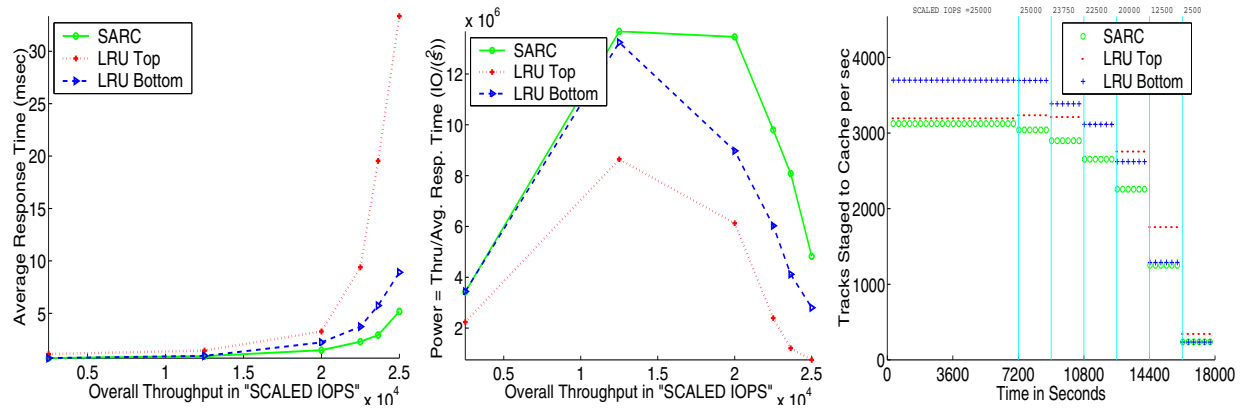
when compared to LRU Top. Hence, LRU Bottom performs better than LRU Top in the cache-sensitive configuration where RANDOM list has more utility, whereas, the reverse is true in the cache-insensitive configuration. However, in both the cases, SARC significantly and dramatically outperforms both the LRU variants by judiciously and dynamically partitioning the cache space amongst the two lists. Due to its self-tuning nature, SARC achieves this without any *a priori* knowledge of the different workloads resulting from different configurations and load levels. For the cache-sensitive configuration (resp. cache-insensitive), at the peak throughput, the overall average response times for LRU Top, LRU Bottom, and, SARC are, respectively, 33.35ms, 8.92ms, and 5.18ms (resp. 8.62ms, 15.26ms, and 6.87ms).

To facilitate a more detailed analysis of the performance improvements due to SARC, Table II provides the break-up of overall average response time into read and write components. At the peak throughput in the cache-sensitive configuration, SARC provides 83.7% and 39.0% read response time reduction over LRU Top and LRU Bottom, respectively. Even for the cache-insensitive configuration at the peak throughput, SARC provides 16.1% and 46.9% read response time reduction over LRU Top and LRU Bottom, respectively.

SARC improves read response times directly by reducing the misses, serendipitously, the resultant reduction in the back-end load also improves the performance for the concurrent writes. At the peak throughput in the cache-sensitive configuration, SARC provides 85.2% and 44.8% write response time reduction over LRU Top and LRU Bottom, respectively. Once again, even for the cache-insensitive configuration at peak throughput, SARC provides 28.6% and 66.7% write response time reduction over LRU Top and LRU Bottom, respectively. Also observe in Table II that although none of the LRU strategies works well in both cache-sensitive and cache-insensitive configurations, SARC outperforms the better of the two LRU variants fairly consistently across all load levels for both reads and writes.

2) *Power of the Storage Controller:* We now provide an alternate viewpoint for studying the throughput versus average response time curves. Typically, at a low (resp. high) throughput one observes a low (resp. high) response time. Thus, there is a trade-off between the two quantities. [42] combined them into a single performance measure *power* that is defined as the ratio of throughput to average response time. The two plots in the middle column of Figure 6 display overall throughput versus power for the three algorithms. The visualization helps us observe the relative performance of the algorithms even at low load levels, where the throughput-response time plots may seem to overlap. In

High Load Schedule with Cache-sensitive back-end configuration



Low Load Schedule with Cache-insensitive back-end configuration

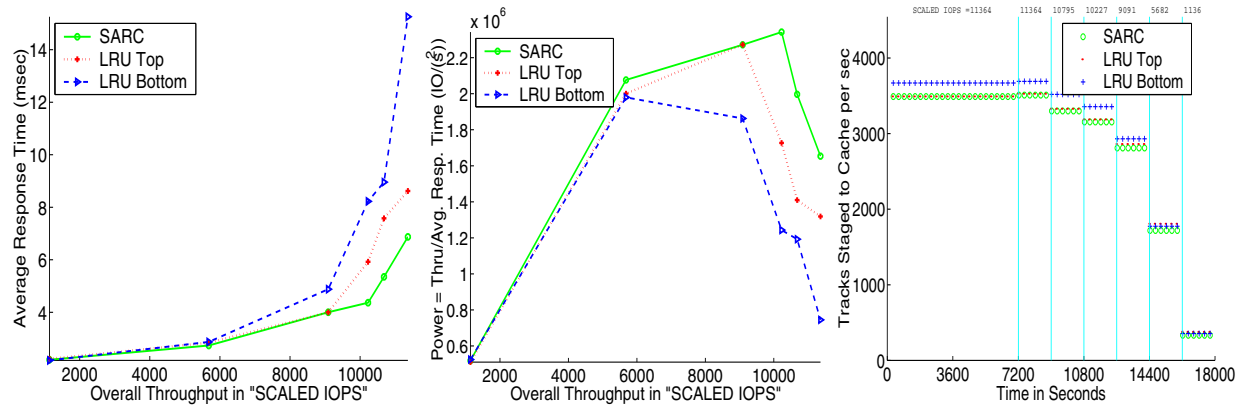


Fig. 6. A comparison of LRU Top, LRU Bottom, and SARC. The top (resp. bottom) three panels correspond to SPC-1 Like in cache-sensitive (resp. cache-insensitive) configuration. For both the configurations, the left column displays throughput versus overall average response times, the middle column displays throughput versus power, and the right column displays the evolution of the rate of tracks staged to cache. The vertical lines demarcate the load schedules in Table I.

Low Load Schedule with Cache-insensitive back-end configuration

Scaled IOPS	LRU-Top read/write	LRU-Bottom read/write	SARC read/write
1136	5.30/ 0.18	5.15/0.18	5.21/ 0.18
5682	6.71/ 0.26	6.77/0.27	6.45/0.26
9091	8.62/ 0.92	9.74/1.64	8.58/0.95
10227	11.13/2.45	13.98/4.39	8.86/1.37
10795	13.23/3.81	14.79/5.07	10.15/2.15
11364	14.36/4.79	22.72/10.28	12.05/3.42

High Load Schedule with Cache-sensitive back-end configuration

Scaled IOPS	LRU-Top read/write	LRU-Bottom read/write	SARC read/write
2500	2.51/ 0.19	1.53/0.19	1.54/ 0.19
12500	3.00/0.41	1.79/0.38	1.76/0.35
20000	5.22/1.96	3.20/1.58	2.38/0.89
22500	12.71/7.18	4.95/2.92	3.34/1.60
23750	24.84/16.00	7.32/4.69	4.06/2.17
25000	41.70/27.78	11.10/7.46	6.77/4.12

TABLE II. A comparison of average read/write response times for LRU Top, LRU Bottom, and SARC. The left (resp. right) table corresponds to cache-insensitive (resp. cache-sensitive) configuration. All the response time numbers are in ms. The best numbers for each load point are in bold.

both configurations, the power of the storage controller with the SARC algorithm envelopes the power from the other algorithms.

3) *Rate of Stages to Cache*: The two plots in the right column of Figure 6 display the rate (per second) of tracks being brought (or staged) to the cache in response to read misses or sequential prefetches. In the cache-sensitive configuration (top right), we observe that LRU Top is better than LRU Bottom for the higher load levels, while the opposite is true for the lighter load levels. In contrast, SARC is consistently better than both the LRU variants for all load levels.

To understand the importance of this metric, note that from a client's perspective, in the cache-sensitive configuration, as seen in the top, left panel of Figure 6, LRU Bottom consistently outperforms LRU Top by delivering a lower average response time. However, from a storage controller's perspective, LRU Bottom outperforms LRU Top for lower loads while the converse is true for higher loads. In contrast, SARC consistently outperforms both the LRU variants from both the perspectives. In other words, not only does SARC deliver a better performance to a client, it does so without unduly stressing the server.

Similar observations also hold for the cache-insensitive configuration, albeit, to a smaller extent.

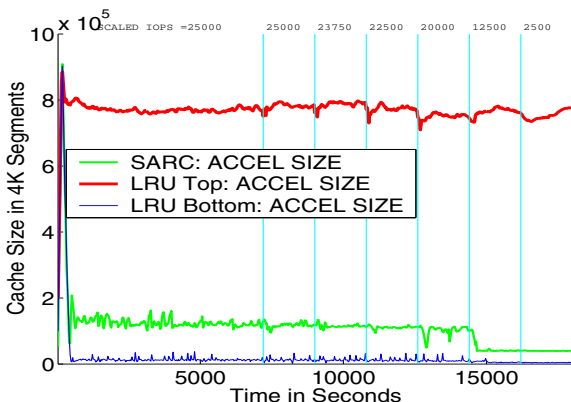


Fig. 7. A comparison of cache space allocated to SEQ list by LRU Top, LRU Bottom, and SARC for the SPC-1 Like workload in cache-sensitive configuration. The vertical lines demarcate the high load schedule in Table I.

4) *Adaptive nature of SARC*: The Figure 7 plots the sizes of the SEQ list versus the time (in seconds) for all the three algorithms. It can be seen that LRU Top allocates too much cache space to SEQ list, LRU Bottom allocates too little cache space to SEQ list, while SARC adaptively allocates just the right amount of cache space to SEQ so as improve the overall performance. It can also be seen that SARC adapts to the evolving workload and selects a different size for the SEQ list at different loads.

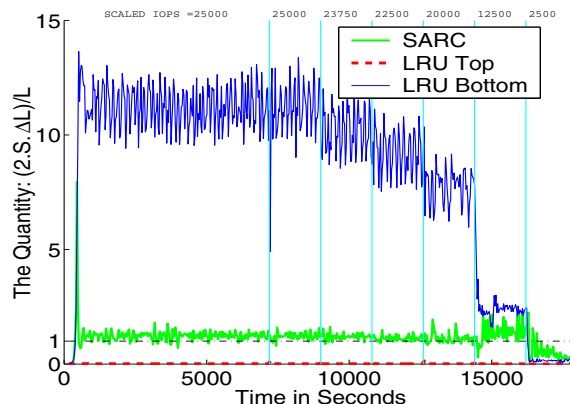


Fig. 8. A comparison of the quantity $(2 \cdot s \cdot \Delta L) / L$ for LRU Top, LRU Bottom, and SARC for the SPC-1 Like workload in cache-sensitive configuration. The vertical lines demarcate the high load schedule in Table I.

The Figure 8 plots the quantity $(2 \cdot s \cdot \Delta L) / L$ for the three algorithms. This quantity is denoted by variable ratio in line 4 of the algorithm in Figure 4. It can be seen that for LRU Bottom keeps ratio too large meaning that it would benefit by further increase in cache space devoted to sequential data. Similarly, LRU Top keeps ratio too small meaning that it would benefit by a decrease in cache space devoted to sequential data. Whereas SARC keeps ratio to roughly the idealized value of 1 meaning that it essentially gets very close to the optimum. In other words, SARC will not benefit by further trading of cache space between SEQ and RANDOM, whereas the other two algorithms would. This plot clearly explains why SARC outperforms the LRU variants in the top three panels of Figure 7. Furthermore, quite importantly, this plot also indicates that any other algorithm that does not directly attempt to drive ratio to 1 as SARC does will not be competitive with SARC. For example, any algorithm that keeps sequential data in the cache for a fixed time [1] or allocates a fixed, constant amount of space to sequential data [29], cannot in general outperform SARC.

Remark IV.1 For heavy sequential workloads, SARC keeps SEQ list just as large enough as useful and does not starve the random workload if present. If there is no random workload then the entire cache will be devoted to the sequential workload and vice versa. When both heavy sequential and heavy random workloads are present, SARC computes the marginal utility to dynamically divide the cache between the two workloads.

V. CONCLUSIONS

We have designed a powerful sequential prefetching strategy that combines virtues of synchronous and

asynchronous prefetching while avoiding the anomaly that arises when prefetching and caching are integrated, and is capable of attaining zero misses for sequential streams.

We have introduced a self-tuning, low overhead, simple to implement, locally adaptive, novel cache management policy **SARC** that dynamically and adaptively partitions the cache space amongst sequential streams and random streams so as to reduce the read misses. **SARC** is doubly adaptive in that it adapts not only the cache space allocated to each class but also the rate at which the cache space is transferred from one class to another. It is extremely easy to convert an existing LRU variant into **SARC**.

We have implemented **SARC** along with two popular state-of-the-art LRU variants on Shark hardware. By using the most widely adopted storage benchmark, we have demonstrated that **SARC** consistently outperforms the LRU variants, and shifts the throughput versus average response time curves to the right thus fundamentally increasing the capacity of the system. Furthermore, **SARC** deliver better performance to a client, without unduly stressing the server.

We believe that the insights, analysis, and algorithm presented in this paper are widely applicable. Due to its adaptivity, we expect **SARC** to work well across (i) a wide range of workloads that may have a varying mix of sequential and random clients and may possess varying temporal locality of the random clients and varying number of sequential and random streams with varying think times; (ii) different back-end storage configurations; and (iii) different data layouts.

ENDNOTES

1. The numbers reported in this paper cannot be used to draw inferences about IBM's products, since (i) Shark (ESS Model 800) does not use any of the above three algorithms; (ii) Shark does not use the sequential prefetching algorithm described above; (iii) our software implementation on Shark hardware is experimental and academic; (iv) we use only one of the two available clusters on Shark; and (v) we have scaled throughput (IOPS) numbers. This paper is not intended to be an official SPC-1 submission, but is an academic study geared to demonstrate that **SARC** is better than LRU variants.

ACKNOWLEDGEMENTS

We are indebted to Michael Benhase, Joseph Hyde, Steven Lowe, and Thomas (Chip) Jarvis for numerous discussions on Shark. We are grateful to Dr. Mustafa Uysal, our shepherd, for detailed comments that greatly improved the readability of the paper.

REFERENCES

- [1] J. Gray and P. J. Shenoy, "Rules of thumb in data engineering," in *ICDE*, pp. 3–12, 2000.
- [2] J. E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
- [3] L. A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Sys. J.*, vol. 5, no. 2, pp. 78–101, 1966.
- [4] M. J. Bach, *The Design of the UNIX Operating System*. Englewood Cliffs, NJ: Prentice-Hall, 1986.
- [5] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*. Prentice-Hall, 1997.
- [6] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Reading, MA: Addison-Wesley, 1995.
- [7] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. FAST Conf.*, pp. 115–130, 2003.
- [8] N. Megiddo and D. S. Modha, "Outperforming LRU with an adaptive replacement cache algorithm," *IEEE Computer*, vol. 37, no. 4, pp. 58–65, 2004.
- [9] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, 2000.
- [10] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach," in *USENIX Summer*, pp. 197–207, 1994.
- [11] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical prefetching via data compression," in *SIGMOD Conference*, pp. 257–266, 1993.
- [12] M. L. Palmer and S. Zdonik, "Fido: A cache that learns to fetch," in *Proc. VLDB Conf.*, Sep 1991.
- [13] T. M. Kroeger and D. D. E. Long, "Design and implementation of a predictive file prefetching algorithm," in *USENIX Annual Technical Conference*, pp. 105–118, 2001.
- [14] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proc. SOSP Conf.*, December 1995.
- [15] F. W. Chang and G. A. Gibson, "Automatic I/O hint generation through speculative execution," in *Operating Systems Design and Implementation*, pp. 1–14, 1999.
- [16] R. J. Feiertag and E. I. Organisk, "The Multics input/output system," in *Proc. 3rd SOSP*, 1971.
- [17] J. Rodriguez-Rosell, "Empirical data reference behavior in data base systems," *IEEE Computer*, vol. 9, pp. 9–13, November 1976.
- [18] P. Hawthorn and M. Stonebraker, "Performance analysis of a relational data base management system," in *Proc. SIGMOD Conf.*, pp. 1–12, May 1979.
- [19] B. T. Zivkov and A. J. Smith, "Disk cache design and performance as evaluated in large timesharing and database," in *Proc. Comput. Measurement Group Conf.*, pp. 639–658, Dec 1997.
- [20] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Trans. Database Systems*, vol. 3, no. 3, pp. 223–247, 1978.
- [21] W. W. Hsu, A. J. Smith, and H. C. Young, "I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level," *ACM Trans. Database Syst.*, vol. 26, no. 1, pp. 96–143, 2001.
- [22] W. W. Hsu, A. J. Smith, and H. C. Young, "Characteristics of production database workloads and the TPC benchmarks," *IBM Sys. J.*, vol. 40, no. 3, pp. 781–802, 2001.
- [23] B. McNutt and S. Johnson, "A standard test of I/O cache," in *Proc. Comput. Measurements Group's 2001 Int. Conf.*, 2001.
- [24] S. A. Johnson, B. McNutt, and R. Reich, "The making of a standard benchmark for open system storage," *J. Comput. Resource Management*, no. 101, pp. 26–32, Winter 2001.
- [25] Storage Performance Council, "SPC Benchmark-2: Public Review Draft Specification, 0.8.0," November 2003.
- [26] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *ACM Trans. Computer Systems*, vol. 2, pp. 181–197, August 1984.
- [27] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a distributed file system," in *Proc. SOSP Conf.*, pp. 198–212, 1991.

- [28] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. D. Kupfer, and J. G. Thompson, "A trace-driven analysis of the UNIX 4.2 BSD file system," in *Proc. 10th SOSF*, pp. 15–24, 1985.
- [29] J. Z. Teng and R. A. Gumaer, "Managing IBM database 2 buffers to maximize performance," *IBM Sys. J.*, vol. 23, no. 2, pp. 211–218, 1984.
- [30] P. Mead, "Oracle Rdb buffer management, www.oracle.com/technology/products/rdb/pdf/2002_tech_forums/rdbtf_2002_buffer.pdf," 2002.
- [31] G. Castets, P. Crowhurst, S. Garraway, and G. Rebmann, "IBM TotalStorage Enterprise Storage Server Model 800." IBM Redbook, October 2002.
- [32] B. C. Beardsley, M. T. Benhase, J. S. Hyde, T. C. Jarvis, D. A. Martin, and R. L. Morton, "Method and system for staging data into cache." US Patent 06381677, issued on April 30, 2002.
- [33] E. A. M. Shriver, A. Merchant, and J. Wilkes, "An analytic behavior model for disk drives with readahead caches and request reordering," in *SIGMETRICS*, pp. 182–191, 1998.
- [34] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," in *SIGMETRICS*, pp. 188–197, 1995.
- [35] N. Ragaz and J. Rodriguez-Rosell, "Empirical studies of storage management in a data base system," tech. rep., RJ 1834, IBM Research Labs, San Jose, October 1976.
- [36] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references," in *Proc. OSDI Conf.*, pp. 119–134, 2000.
- [37] B. C. Beardsley, M. T. Benhase, D. A. Martin, R. L. Morton, and M. A. Reid, "Data management method in cache, involves selecting one of least recently used data entries for demoting it during new data entry." US Patent 6141731, filed on August 19, 1998, issued on October 10, 2000.
- [38] S. F. Kaplan, L. A. McGeoch, and M. F. Cole, "Adaptive caching for demand prepagging," in *MSP/ISMM*, pp. 221–232, 2002.
- [39] M. H. Hartung, "IBM TotalStorage Enterprise Storage Server: A designer's view," *IBM Sys. J.*, vol. 42, no. 2, pp. 383–396, 2003.
- [40] V. Sundaram, P. Goyal, P. Radkov, and P. Shenoy, "Evaluation of object placement techniques in a policy-managed storage system," tech. rep., TR03-38, Dept. Comput. Sci., Univ. Mass., Nov 2003.
- [41] D. Martin, "Configuring the IBM Enterprise Storage Server for Oracle OLTP applications," tech. rep., IBM, April 2003.
- [42] L. Kleinrock, "On flow control in computer networks," in *Proc. Int'l. Conf. Commun., Toronto, Ontario, Canada*, vol. II, pp. 27.2.1–27.2.5, June 1978.

SLINKY: Static Linking Reloaded

Christian Collberg, John H. Hartman, Sridivya Babu, Sharath K. Udupa

Department of Computer Science

University of Arizona

Tucson, AZ 85721

{collberg,jhh,divya,sku}@cs.arizona.edu

Abstract

Static linking has many advantages over dynamic linking. It is simple to understand, implement, and use. It ensures that an executable is self-contained and does not depend on a particular set of libraries during execution. As a consequence, the user executes exactly the same executable image as was tested by the developer, diminishing the risk that the user's environment will affect correct behavior.

The major disadvantages of static linking are increases in the memory required to run an executable, network bandwidth to transfer it, and disk space to store it.

In this paper we describe the SLINKY system that uses digest-based sharing to combine the simplicity of static linking with the space savings of dynamic linking: although SLINKY executables are completely self-contained, minimal performance and disk-space penalties are incurred if two executables use the same library. We have developed a SLINKY prototype that consists of tools for adding digests to executables, a slight modification of the Linux kernel to use those digests to share code pages, and tools for transferring files between machines based on digests of their contents. Results show that our prototype has no measurable performance decrease relative to dynamic linking, a comparable memory footprint, a 20% storage space increase, and a 34% increase in the network bandwidth required to transfer the packages. We believe that SLINKY obviates many of the justifications for dynamic linking, making static linking a superior technology for software organization and distribution.

1 Introduction

Most naïve users' frustrations with computers can be summarized by the following two statements: "I installed this new program and it just didn't work!" or "I downloaded a cool new game and suddenly this

other program I've been using for months stopped working!" In many cases these problems can be traced back to missing, out-of-date, or incompatible dynamic libraries on the user's computer. While this problem may occur in any system that supports dynamically linked libraries, in the Windows community it is affectionately known as *DLL Hell* [12].

In this paper we will argue — against conventional wisdom — that in most cases dynamic linking should be abandoned in favor of static linking. Since static linking ensures that programs are self-contained, users and developers can be assured that a program that was compiled, linked, and tested on the developer's machine will run unadulterated on the user's machine. From a quality assurance and release management point of view this has tremendous advantages: since a single, self-contained, binary image is being shipped, little attention must be made to potential interference with existing software on the user's machine. From a user's point of view there is no chance of having to download additional libraries to make the new program work.

A major argument against static linking is the size of the resulting executables. We believe that this will likely be offset by ever-increasing improvements in disk-space, CPU speed, and main memory size. Nonetheless, this paper will show that the cost of static linking, in terms of file-system storage, network bandwidth, and run-time memory usage, can be largely eliminated through minor modifications to the operating system kernel and some system software.

Our basic technique is *digest-based sharing*, in which *message digests* identify identical chunks of data that can be shared between executables. Digests ensure that a particular chunk of data is only stored once in memory or on disk, and only transported once over the network. This sharing happens without intervention by the software developer or system administrator; SLINKY automatically finds and shares identical chunks of different executables.

1.1 SLINKY

SLINKY is a relatively simple system that provides the space-saving benefits of dynamic linking without the complexities. There are four components to SLINKY:

1. The `slink` program creates a statically-linked executable from one that is dynamically linked. It does so by linking in the dynamic libraries on which the program depends, resolving references, and performing necessary page alignment.
2. The `digest` program adds SHA-1 digests of the code pages to the statically-linked executable produced by `slink`.
3. A modified Linux kernel shares code pages between processes based on the digests added by `digest`. During a page fault the digest of the faulting page is used to share an in-memory copy of the page, if one exists.
4. The `ckget` program downloads software packages based on the digests of variable-sized chunks. Chunk boundaries are computed using Rabin fingerprints. Each unique chunk is only downloaded once, greatly reducing the network bandwidth required to transfer statically-linked executables.

What is particularly attractive about SLINKY is its simplicity. The `slink` program consists of 1000 lines of code, `digest` 200 lines, the kernel modifications 100 lines, and `ckget` 100 lines.

1.2 Background

“Linking” refers to combining a program and its libraries into a single executable and resolving the symbolic names of variables and functions into their resulting addresses. Generally speaking, *static linking* refers to doing this process at link-time during program development, and incorporating into each executable the libraries it needs. *Dynamic linking* refers to deferring linking until the executable is actually run. An executable simply contains references to the libraries it uses, rather than copies of the libraries, dramatically reducing its size. As a result, only a single copy of each library must be stored on disk. Dynamic linking also makes it possible for changes to a library to propagate automatically to the executables that use it, since they will be linked against the new version of the library the next time they are run.

Dynamic linking is typically used in conjunction with *shared libraries*, which are libraries whose in-memory images can be shared between multiple processes. The combination of dynamic linking and shared libraries ensures that there is only one copy of a shared library on disk and in memory, regardless of how many executables make use of it.

Dynamic linking is currently the dominant practice, but this was not always the case. Early operating systems used static linking. Systems such as Multics [3] soon introduced dynamic linking as a way of saving storage and memory space, and increasing flexibility. The resulting complexity was problematic, and subsequent systems such as Unix [16] went back to static linking because of its simplicity. The pendulum has since swung the other way and dynamic linking has become the standard practice in Unix and Linux.

1.3 DLL Hell

Although dynamic linking is the standard practice, it introduces a host of complexities that static linking does not have. In short, running a dynamically linked executable depends on having the proper versions of the proper libraries installed in the proper locations on the computer. Tools have been developed to handle this complexity, but users are all too familiar with the “DLL Hell” [12] that can occur in ensuring that all library dependencies are met, especially when different executables depend on different versions of the same library.

DLL Hell commonly occurred in early versions of the Windows operating when program installation caused an older version of a library to replace an already installed newer one. Programs that relied on the newer version then stopped working — often without apparent reason. Unix and newer versions of Windows fix this problem using complex versioning schemes in which the library file name contains the version number. The numbering scheme typically allows for distinction between compatible and incompatible library versions. However, DLL Hell can still occur if, for example, the user makes a minor change to a library search `PATH` variable, or compatible library versions turn out not to be. Sometimes programs depend on undocumented features (or even bugs) in libraries, making it difficult to determine when two versions are compatible. No matter what the cause, DLL Hell can cause a previously working program to fail.

1.4 Anecdotes

It is easy to argue that the “right solution” to the DLL Hell problem is that *“every package maintainer should just make sure that their package always has the correct set of dependencies!”*, perhaps by using dependency management tools. However, it appears that dependency problems still occur even in the most well-designed package management systems, such as Debian’s `apt` [2]. Below we present some anecdotal evidence of this.

Consider, first, the following instructions from the Debian quick reference [1] guide:

```
Package dependency problems may occur
when upgrading in unstable/testing.
Most of the time, this is because a
package that will be upgraded has a
new dependency that isn't met. These
problems are fixed by using
```

```
# apt-get dist-upgrade
```

```
If this does not work, then repeat
one of the following until the problem
resolves itself [sic]:
```

```
# apt-get upgrade -f
```

```
... Some really broken upgrade
scripts may cause persistent trouble.
```

Most Debian users will install packages from the `unstable` distribution, since the `stable` distribution is out-of-date.

Similarly, searching Google with the query “`apt-get dependency problem`”, produces many hits of the following kind:

```
igor> I am trying to upgrade my potato
igor> distro to get the latest KDE. after I
igor> type apt-get update && apt-get
igor> upgrade, here is my error message:
igor> -----
igor> ...
igor> Sorry, but the following packages
igor> have unmet dependencies:
igor>  debianutils: PreDepends: libc6
igor>    (>= 2.1.97) but 2.1.3-18 is to
igor>    be installed
igor> E: Internal Error, InstallPackages
igor> was called with broken packages!
igor> -----
briand> I'm getting everything from the
briand> site you are using and everything
briand> is OK. This is very odd. ...
```

Ardour is a 200,000 LOC multi-track recording tool for Linux, which relies on many external libraries for everything from GUI to audio format conversion. Its author, Paul Davis, writes [4]:

I spent a year fielding endless reports from people about crashes, strange backtraces and the like before giving up and switching to static linking. Dynamic linking just doesn't work reliably enough.

I will firmly and publically denounce any packaging of Ardour that use dynamic linking to any C++ library except (and possibly including) `libstdc++`.

1.5 Contributions

In general, package maintenance and installation are difficult problems. A software distribution that appears to work fine on the package maintainer's machine may break for any number of reasons on the user's machine. The inherent complexity of dynamic library versioning coupled with the fact that an installation package has to work under any number of operating system versions and on computers with any number of other packages installed, almost invariably invites a visit from Murphy's famous law.

We have developed a system called SLINKY that combines the advantages of static and dynamic linking without the disadvantages of either. In our system, executables are statically linked (and hence avoid the complexities and frailties of dynamically linked programs) but can be executed, transported, and stored as efficiently as their dynamic counterparts.

The key insight is that dynamic linking saves space by explicitly sharing libraries stored in separate files, and this introduces much of the complexity. SLINKY, instead, relies on implicit sharing of identical chunks of data, which we call *digest-based sharing*. In this scheme chunks of data are identified by a *message digest*, which is a cryptographically secure hash such as SHA-1 [19]. Digest-based sharing allows SLINKY to store a single copy of each data chunk in memory and on disk, regardless of how many executables share that data. The digests are also used to transfer only a single copy of data across the network. This technique allows SLINKY to approach the space savings of dynamic linking without the complexity.

The rest of the paper is organized as follows. In Section 2 we compare static and dynamic linking. In Section 3 we describe the implementation of the SLINKY system. In Section 4 we show that empirically our system makes static linking as efficient as dynamic linking. In Section 5 we discuss related work. In Section 6, finally, we summarize our results.

2 Linking

High-level languages use symbolic names for variables and functions such as `foo` and `read()`. These *symbols* must be converted into the low-level addresses understood by a computer through a process called *linking* or *link editing* [9]. Generally, linking involves assigning data and instructions locations in the executable's address space, determining the resulting addresses for all symbols, and *resolving* each symbol reference by replacing it with the symbol's address. This process is somewhat complicated by *libraries*, which are files containing commonly-used variables and functions. There are many linking variations, but they fall into two major categories, *static linking* and *dynamic linking*.

2.1 Static Linking

Static linking is done at *link-time*, which is during program development. The developer specifies the libraries on which a executable depends, and where to find them. A tool called the *linker* uses this information to find the proper libraries and resolve the symbols. The linker produces a static executable with no unresolved symbols, making it self-contained with respect to libraries.

A statically-linked executable may be self-contained, but it contains portions of each library to which it is linked. For large, popular libraries, such as the C library, the amount of duplicate content can be significant. This means that the executables require more disk storage, memory space, and network bandwidth than if the duplicate content were eliminated.

Statically linking executables also makes it difficult to update libraries. A statically-linked executable can only take advantage of an updated library if it is relinked. That means that the developer must get a new copy of the library, relink the executable and verify that it works correctly with the new library, then redistribute it to the users. These drawbacks with statically-linked executables led to the development of dynamic linking.

2.2 Dynamic Linking

Dynamic linking solves these problems by deferring symbol resolution until the executable is run (*run-time*), and by using a special library format (called a *dynamic library* or *shared library*) that allows processes to share a single copy of a library in memory. At link-time all the linker does is store the names of the necessary libraries in the executable. When

the executable runs, a program called a *dynamic linker/loader* reads the library names from the executable and finds the proper files using a list of directories specified by a library search path. Since the libraries aren't linked until run-time, the executable may run with different library versions than were used during development. This is useful for updating a library, because it means the executables that are linked to that library will use the new version the next time they are run.

While the "instant update" feature of dynamic linking appears useful — it allows us to fix a bug in a library, ship that library, and instantly all executables will make use of the new version — it can also have dangerous consequences. There is a high risk of programs failing in unpredictable ways when a new version of a library is installed. Even though two versions of a library may have compatible interfaces and specifications, programs may depend on undocumented side-effects, underspecified portions of the interface, or other features of the library, including bugs. In addition, if a library bug was known to the developer he may have devised a "work-around", possibly in a way no longer compatible with the bug fix. In general, it is impossible for a library developer to determine when two versions are compatible since the dependencies on the library's behavior are not known. We believe that *any* change to a library (particularly one that has security implications) requires complete regression tests to be re-run for the programs that use the library.

Dynamic linking also complicates software removal. Care must be taken to ensure that a dynamic library is no longer in use before deleting it, otherwise executables will mysteriously fail. On the other hand, deleting executables can leave unused dynamic libraries scattered around the system. Either way, deleting a dynamically-linked executable isn't as simple as deleting the executable file itself.

2.3 Code-Sharing Techniques

Systems that use dynamic linking typically share a single in-memory copy of a dynamic library among all processes that are linked to it. It may seem trivial to share a single copy, but keep in mind that a library will itself contain symbol references. Since each executable is linked and run independently, a symbol may have a different address in different processes, which means that shared library code cannot contain absolute addresses.

The solution is *position-independent code*, which is code that only contains relative addresses. Absolute addresses are stored in a per-process *indi-*

rection table. Position-independent code expresses all addresses as relative offsets from a register. A dedicated register holds the base address of the indirection table, and the code accesses the symbol addresses stored in the table using a relative offset from the base register. The offsets are the same across all processes, but the registers and indirection tables are not. Since the code does not contain any absolute addresses it can be shared between processes. This is somewhat complex and inefficient, but it allows multiple processes to share a single copy of the library code.

2.4 Package Management

The additional flexibility dynamic linking provides also introduces a tremendous amount of complexity. First, in order to run an executable the libraries on which it depends must be installed in the proper locations in the dynamic linker/loader's library search path. This means that to run an executable a user needs both the executable and the libraries on which it depends, and must ensure that the dynamic linker is configured such that the libraries are on the search path. Additionally, the versions of those libraries must be compatible with the version that was used to develop the executable. If they are not, then the executable will either fail to run or produce erroneous results.

To manage this complexity, package systems such as RedHat's `rpm` [17] and Debian's `dpkg` [5] were developed. A package contains everything necessary to install an executable or library, including a list of the packages on which it depends. For an executable these other packages include the dynamic libraries it needs. A sophisticated versioning scheme allows a library package to be updated with a compatible version. For example, the major number of a library differentiates incompatible versions, while the minor number differentiates compatible versions. In this way a package can express its dependency on a compatible version of another package. The versioning system must also extend to the library names, so that multiple versions of the same library can coexist on the same system.

The basic package mechanism expresses inter-package dependencies, but it does nothing to resolve those dependencies. Suppose a developer sends a user a package that depends on another package that the user does not have. The user is now forced to ask for the additional package, or search the Internet looking for the needed package. More recently, the user could employ sophisticated tools such as RedHat's `up2date` [21] or Debian's `apt` [2]

to fetch and install the desired packages from on-line repositories.

2.5 Security Issues

Dynamic linking creates potential security holes because of the dependencies between executables and the libraries they need. First, an exploit in a dynamic library affects every executable that uses that library. This makes dynamic libraries particularly good targets for attack. Second, an exploit in the dynamic linker/loader affects every dynamically-linked executable. Third, maliciously changing the library search path can cause the dynamic linker/loader to load a subverted library. Fourth, when using a package tool such as `up2date` or `apt`, care must be taken to ensure the authenticity and integrity of the downloaded packages. These potential security holes must be weighed against the oft-stated benefit that dynamic linking allows for swift propagation of security fixes.

3 SLINKY

SLINKY is a system that uses message digests to share data between executables, rather than explicitly sharing libraries. Chunks of data are identified by their digest, and SLINKY stores a single copy of each chunk in memory and disk. SLINKY uses SHA-1 [19] to compute digests. SHA-1 is a hash algorithm that produces a 160-bit value from a chunk of data. SHA-1 is cryptographically secure, meaning (among other things) that although it is relatively easy to compute the hash of a chunk of data, it is computationally infeasible to compute a chunk of data that has a given hash. There are no known instances of two chunks of data having the same SHA-1 hash, and no known method of creating two chunks that have the same hash. This means that for all practical purposes chunks of data with the same SHA-1 hash are identical. Although there is some controversy surrounding the use of digests to identify data [7], it is generally accepted that a properly-designed digest function such as SHA-1 has a negligible chance of hashing two different chunks to the same value. It is much more likely that a hardware or software failure will corrupt a chunk's content.

Depending on one's level of paranoia with respect to the possibility of hash collisions, different hashing algorithms could be used. The increased costs come in both time and space and depend on the size of the hash (128 bits for MD5; 160 bits for SHA-1; 256, 384, or 512 bits for newer members of the SHA family) and the number of rounds and complexity

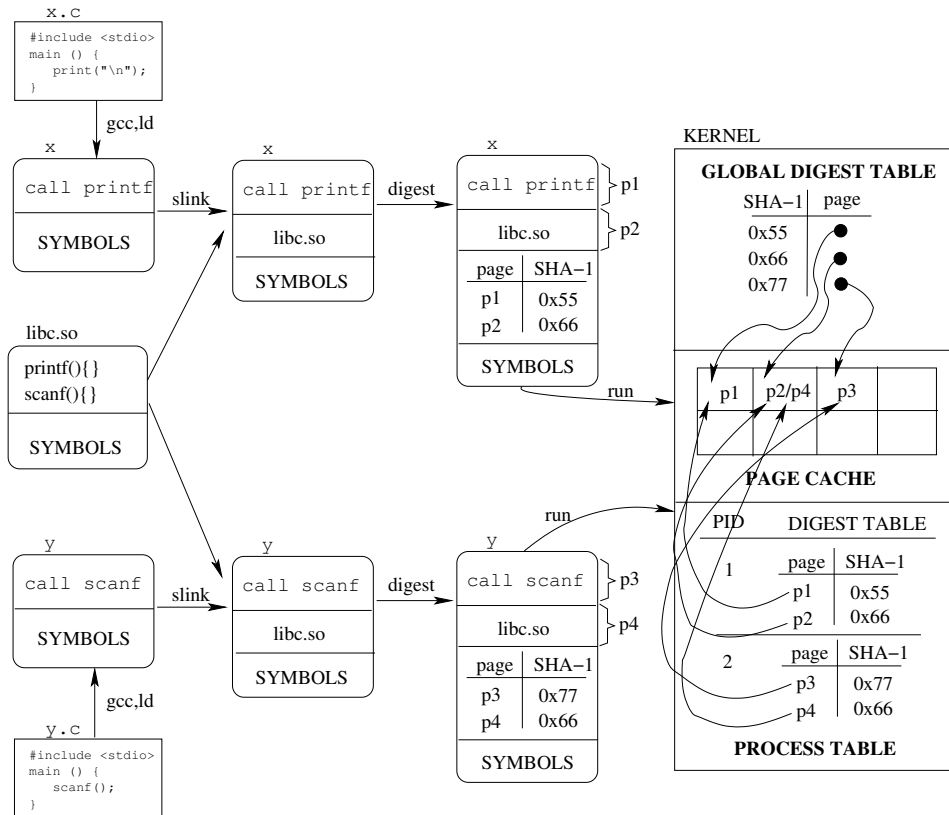


Figure 1: Sharing memory pages based on digest. Only one copy of each page exists in memory (page cache). The per-process digest table contains the page digests of the process, and is initialized from a table in the executable. The global digest table contains the digest of every page in memory.

of computation. Schneier [18, p. 456] shows, for example, that MD5 is roughly 2.3 times faster than SHA-1, while SHA-1 is less prone to collisions due to its larger hash-size.

Digests allow SLINKY to store a single copy of each chunk in memory and on disk. In addition, when transferring an executable over the network only those chunks that do not already exist on the receiving end need be sent. The use of digests allows SLINKY to share data between executables efficiently, without introducing the complexities of dynamic linking. In addition, SLINKY avoids the security holes endemic to dynamic linking.

The following sections describe how SLINKY uses digests to share data in memory and on disk, as well as to reduce the amount of data required to transfer an executable over the network. We developed a SLINKY prototype that uses digests to share memory pages and eliminate redundant network transfers. Sharing disk space based on digest is currently work-in-progress; we describe how SLINKY will pro-

vide that functionality, although it has not yet been implemented.

3.1 Sharing Memory Pages

SLINKY shares pages between processes by computing the digest of each code page, and sharing pages that have the same digest. If a process modifies a page, then the page's digest changes, and the page can no longer be shared with other processes using the old version of the page. One way to support this is to share the pages copy-on-write. When a process modifies a page it gets its own copy. SLINKY employs the simpler approach of only sharing read-only code pages. SLINKY assumes that data pages are likely to be written, and therefore unlikely to be shared anyway.

The current SLINKY prototype is Linux-based, and consists of three components that allow processes to share pages based on digests. The first is a linker called `slink` that converts a dynamically-

linked executable into a statically-linked executable. The second is a tool called **digest** that computes the digest of each code page in an executable. The digests are stored in special sections in the executable. The third component is a set of Linux kernel modifications that allow processes to share pages with identical digests.

Figure 1 illustrates how SLINKY functions. A source file `x.c` that makes use of the C library is compiled into a dynamically-linked executable file `x`. Our program **slink** links `x` with the dynamic library `libc.so`, copying its contents into `x`. The **digest** program adds a new ELF section that maps pages to their digests. The process is repeated for a second example program `y.c`. When `x` is run its pages will be loaded, along with their digests. When `y` is run, page `p4` will not be loaded since a page with the same digest already exists (page `p2` from `x`'s copy of `libc.so`).

3.1.1 Slink

Shared libraries require position-independent code to allow sharing of code pages between processes. SLINKY also requires position-independent code because static linking may cause the same library to appear at different addresses in different executables. Therefore, SLINKY must statically link executables with shared libraries, since those libraries contain position-independent code and traditional static libraries do not. **Slink** is a program that does just that — it converts a dynamically-linked executable into a statically-linked executable by linking in the shared libraries on which the dynamically-linked executable depends. The resulting executable is statically linked, but contains position-independent code from the shared libraries. **Slink** consists of about 830 lines of C and 160 lines of shell script.

The input to **slink** is a dynamically-linked executable in ELF format [10]. **Slink** uses the **prelink** [13] tool to find the libraries on which the executable depends, organize them in the process's address space, and resolve symbols. **Slink** then combines the prelinked executable and its libraries into a statically-linked ELF file, aligning addresses properly, performing some relocations that **prelink** cannot do, and removing data structures related to dynamic linking that are no longer needed.

Although position-independent code is necessary for sharing library pages between executables, it is not sufficient. If the same library is linked into different executables at different page alignments, the page contents will differ and therefore cannot be

shared. Fortunately, the page alignment is specified in the library ELF file, ensuring that all copies of the library are linked with the same alignment.

3.1.2 Digest

Digest is a tool that takes the output from **slink** and inserts the digests for each code page. For every executable read-only ELF segment, **digest** computes the SHA-1 hash of each page in that segment and stores them in a new ELF section. This section is indexed by page offset within the associated segment, and is used by the kernel to share pages between processes. A Linux page is 4KB, and the digest is 20 bytes, so the digests introduce an overhead of 20/4096 or less than 0.5% per code page. **Digest** consists of about 200 lines of C code.

3.1.3 Kernel Modifications

SLINKY requires kernel modifications so that the loader and page fault handler make use of the digests inserted by **digest** to share pages between processes. These modifications consist of about 100 lines of C. When a program is loaded, the loader reads the digests from the file and inserts them in a per-process digest table (PDT) that maps page number to digest. This table is used during a page fault to determine the digest of the faulting page.

We also modified the Linux 2.4.21 kernel to maintain a global digest table (GDT) that contains the digest of every code page currently in memory. The GDT is used during a page fault to determine if there is already a copy of the faulting page in memory. If not, the page is read into memory from disk and an entry added to the GDT. Otherwise the reference count for the page is simply incremented. The page table for the process is then updated to refer to the page, and the process resumes execution. When a process exits the reference count for each of its in-memory pages is decremented; a page is removed from the GDT when its reference count drops to zero.

3.1.4 Security

SLINKY shares pages based on the digests stored in executables, so the system correctness and security depend on those digests being correct. A malicious user could modify a digest or a page so that the digest no longer corresponds to the page's contents, potentially causing executables to use the wrong pages. SLINKY avoids this problem by verifying the digest of each text page when it is brought into memory and before it is added to the GDT.

This slows down the the handling of page faults that result in a page being read from disk, but we show in Section 4.1.2 that this overhead is negligible when compared to the cost of the disk access. Once a page is added to the GDT, SLINKY relies on the memory protection hardware to ensure that processes cannot modify the read-only text page.

Figure 2 shows pseudo-code for the modified page fault handler that is responsible for searching the GDT for desired pages, and adding missing pages to the GDT when they are read from disk.

3.2 Sharing Disk Space

Digests can also reduce the disk space required to store statically-linked executables. One option is to store data based on the per-page digests stored in the executable. Although this will reduce the space required, it is possible to do better. This is because the digests only refer to the executable's code pages, and the virtual memory system requires those pages be fixed-size and aligned within the file. Additional sharing is possible if arbitrary-size chunks of unaligned data are considered.

SLINKY shares disk space between executables by breaking them into variable-size chunks using Rabin's fingerprinting algorithm [15], and then computing the digests of the chunks. Only one copy of each unique chunk is stored on disk. The technique is based on that used in the Low-Bandwidth File system (LBFS) [11]. A small window is moved across the data and the Rabin fingerprint of the window is computed. If the low-order N bits of the fingerprint match a pre-defined value, a chunk boundary is declared. The sliding window technique ensures that the effects of insertions or deletions are localized. If, for example, one file differs from another only by missing some bytes at the beginning, the sliding window approach will synchronize the chunks for the identical parts of the file. Alternatively, if fixed-size blocks were used, the first block of each file would not have the same hash due to the missing bytes, and the mismatch would propagate through the entire file.

SLINKY uses the same 48-byte window as LBFS as this was found to produce good results. SLINKY also uses the same 13 low-order bits of the fingerprint to determine block boundaries, which results in an 8KB average block size. The minimum block size is 2KB and the maximum is 64KB. Using the same parameters as LBFS allows us to build on LBFS's results.

The SLINKY prototype contains tools for breaking files into chunks, computing chunk digests, and

comparing digests between files. It does not yet use the digests to share disk space between executables. We intend to extend SLINKY so that executables share chunks based on digest, but that work is in progress. The current tools allow SLINKY's space requirements to be compared to dynamic libraries, as described in Section 4.

3.3 Reducing Network Bandwidth

The final piece of the puzzle is to reduce the amount of network bandwidth required to transport statically-linked executables. Digests can also be used for this purpose. The general idea is to only transfer those chunks that the destination does not already have. This is the basic idea behind LBFS, and SLINKY uses a similar mechanism. Suppose we want to transfer an executable from X to Y over the network. First, X breaks the executable into chunks and computes the digests of the chunks. X then sends the list of digests to Y. Y compares the provided list with the digests of chunks that it already has, and responds with a list of missing digests. X then sends the missing chunks to Y.

We developed a tool called `ckget` that transfers files across the network based on chunks. Continuing the above example, each file on X has an associated *chunk file* containing the chunk digests for the file. X and Y also maintain individual *chunk databases* that indicate the location of each chunk within their file systems. To transfer a file, Y runs `ckget URL`, which uses the URL to contact X. X responds with the corresponding chunk file. `ckget` cross-references the chunks listed in the file with the contents of its chunk database to determine which chunks it lacks. It then contacts X to get the missing chunks, and reconstitutes the file from its chunks. `ckget` then adds the new chunks to Y's chunk database.

Figure 3 illustrates this process. The client issues the command `ckget x-1.1.deb` to download version 1.1 of the `x` application. `ckget` retrieves the chunk file `x-1.1.ck` from the proper server. The chunk file indicates that two chunks (A and B) are the same as in version 1.0 and already exist locally in `x-1.0.deb`, but a new chunk E needs to be downloaded from the server. `ckget` gets this chunk and then reconstitutes `x-1.1.deb` from `x-1.0.deb` and the downloaded chunk E. `ckget` updates the client's `ChunkDB` to indicate that chunk E now exists in `x-1.1.deb`. Should the user subsequently download `y-2.3.deb` only chunk D will be retrieved.

```

// filemap_nopage is called when the page is not
// recorded in the page table entries of the process
func filemap_nopage (vm, address){
    if (page is in the page cache) return page
    if (page is from a slinky binary) {
        get digest of page from PDT
        use digest to search GDT
        if (matching digest is found) return page
    }
    //page not in memory
    read page from disk
    if (page is from a slinky binary) {
        verify page digest
        add page to GDT
    }
    return page
}

```

Figure 2: Page fault handler. The page's digest is used to search the GDT. If the page is not found it is read from disk, its digest verified, and an entry added to the GDT.

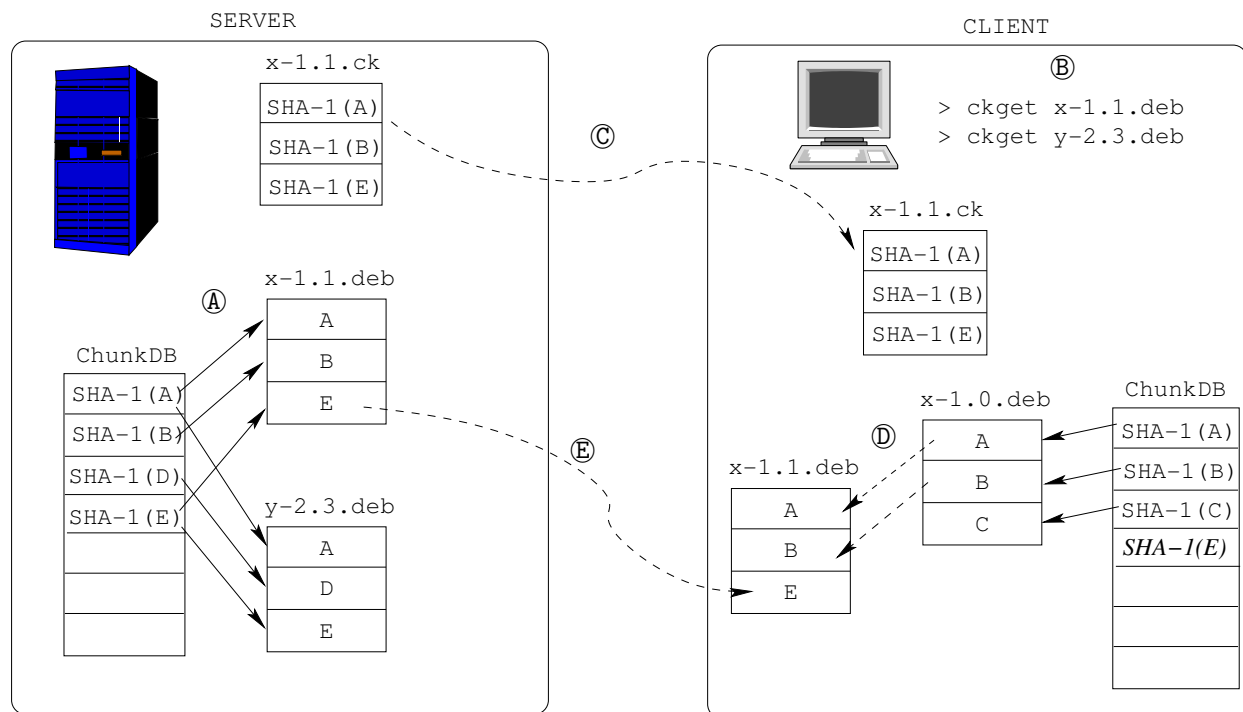


Figure 3: Overview of how SLINKY uses `ckget` to download a software package. Point ① shows the server with two packages `x-1.1.deb` and `y-2.3.deb`. The chunk database maps SHA-1 digests of chunks in these packages to the locations of the chunks. At point ② the client has download of `x-1.1` by issuing the command `ckget x-1.1.deb`. At ③ `x`'s chunk-file is downloaded. At ④ chunks A and B are retrieved from a previous version of `x`. At ⑤ chunk E has to be fetched from the server and its SHA-1 digest added to the client's ChunkDB. If the client subsequently downloads package `y-2.3.deb`, only chunk D will be transferred.

4 Evaluation

We performed several experiments on the SLINKY prototype to evaluate the performance of statically-linked executables vs. dynamically-linked executables, as well as the space required to store them.

4.1 Performance

We performed several experiments to compare the performance of SLINKY with that of a standard dynamically-linked Linux system. First, we measured the elapsed time to build the Linux kernel using `make`. This is representative of SLINKY's performance on real-world workloads. Second, we timed page fault handling in both an unmodified Linux kernel and a kernel modified to support SLINKY executables. SLINKY's CPU overhead is confined to page fault handling, so these timings allow extrapolation of the worst-case overhead for SLINKY when running executables whose performance is limited by page fault performance. Third, we measured the number of page faults when running a variety of executables in both systems. This not only provides information on SLINKY's effect on the page fault rate, but also information on the memory footprint of SLINKY executables.

All experiments were performed on a system with a 2.4 GHz Intel Pentium 4 CPU, 1GB of memory, and a Western Digital WD800BB-53CAA1 80GB disk drive. The kernel was Linux 2.4.21, and the Linux distribution was the "unstable" Debian distribution. The machine was a desktop workstation used for operating system development, so it had a representative set of software development and GUI applications installed. For the SLINKY results all executables used in the tests were created using the `slink` and `digest` tools.

4.1.1 Elapsed Time

Our macro-benchmark consisted of measuring the elapsed time to build the Linux 2.4.19 kernel and several standard drivers. The benchmark was run four times on both the SLINKY and the standard system. There is no statistically-significant difference in performance. The average elapsed time for the standard system was 139.32 ± 0.12 seconds vs. 139.00 ± 0.35 seconds for SLINKY.

4.1.2 Page Fault Handling

We also measured the time required by SLINKY to handle page faults. When a page is not found in the page cache SLINKY looks for it in the GDT, and

when a page is read in from disk SLINKY verifies its digest and adds it to the GDT. Both of these cases increase the page fault overhead. In practice, however, the overhead is minimal. For page faults that require a disk read, SLINKY adds 44.58 microseconds for a total of 3259.55 microseconds per fault, or 1.3%. Page faults that hit in the GDT required 0.95 microseconds. For comparison, page faults that hit in the page cache required 0.82 microseconds.

4.1.3 Memory Footprint

Table 1 shows the results of running several executables on a SLINKY system and a standard system and measuring the number of page faults. For SLINKY page faults are classified into those for which the page was found in the page cache or read from disk (*Other*), vs. those for which the page was found in the GDT (*GDT*). For the standard system page faults are classified into those for shared libraries (*Library*) and those for the executable itself (*Other*). SLINKY adds overhead to both kinds of faults, as described in the previous section. Table 1(a) shows the results of running `make` with an empty GDT. The SLINKY version of `make` generated 95 page faults, compared with 173 faults for the dynamic version. Table 1(b) shows running several commands in sequence starting with an empty GDT. Three conclusions can be drawn from these results. First, for our workloads SLINKY executables suffer about 60-80 fewer page faults than their dynamically-linked counterparts. These additional page faults are caused by the dynamic linker, which is not run for the SLINKY executables.

Second, `make` suffers 95 non-GDT page faults when it is run with an empty GDT, but only 36 non-GDT faults after several other executables have added pages to the GDT. The remaining 59 faults were handled by the GDT. This shows that pages can effectively be shared by digest, without resorting to shared libraries.

Third, the memory footprints for the SLINKY executables are comparable to those for shared libraries. The dynamically-linked version of `make` caused 31 faults to pages not in shared libraries. In comparison, the SLINKY version caused 36 faults when run after several other commands. Therefore, the SLINKY version required 5 more non-shared pages, or 16% more. As more pages are added to the GDT we expect the number for SLINKY to drop to 31. This effect can be seen for `gcc`, which caused 20 page faults in both the SLINKY and dynamic versions, so that both versions had exactly the same memory footprint.

Execution Workload	SLINKY			Dynamic		
	GDT	Other	Sum	Library	Other	Sum
1> make	0	95	95	142	31	173

(a) First workload

Execution Workload	SLINKY			Dynamic		
	GDT	Other	Sum	Library	Other	Sum
1> ls	0	79	79	151	14	165
2> cat	29	3	31	89	3	92
3> gcc	32	20	52	93	20	113
4> make	59	36	95	142	31	173

(b) Second workload

Table 1: The number of page faults generated for two experimental workloads, running SLINKY vs. dynamically-linked executables. The first workload consists of running `make`; the second of running a sequence of commands culminating with `make`. For SLINKY the *GDT* column is faults for which the page was found in the GDT, *Other* is faults that either hit in the page cache or required a disk access, and *Sum* is the total faults. For Dynamic, the *Library* column is faults for pages in shared libraries, *Other* is faults to the executable itself, and *Sum* is the total.

4.2 Storage Space

Table 2(a) shows the space required to store dynamically-linked executables vs. statically-linked. These numbers were collected on the “unstable” Debian distribution. The *Dynamic* column shows the space required to store the dynamically-linked ELF executables in the given directories, plus the dynamic libraries on which they depend. Each dynamic library is only counted once. The *All* row is the union of the directories in the other rows, hence its value is not the sum of the other rows (since libraries are shared between rows). The SLINKY column shows the space required to store the statically-linked executables, and *Ratio* shows the ratio of the static and dynamic sizes. The static executables are much larger than their dynamic counterparts because they include images of all the libraries they use.

Table 2(b) shows the amount of space required to store the dynamic and static executables if they are broken into variable-size chunks and only one copy of each unique chunk is stored. The dynamic executables show a modest improvement over the numbers in the previous table due to commonality in the files. The static executables, however, show a tremendous reduction in the amount of space. Most of the extra space in Table 2(a) was due to duplicate libraries; the chunk-and-digest technique is able to share these chunks between executables. The SLINKY space requirements are reasonable – across

all directories SLINKY consumes 20% more space than dynamic linking. SLINKY requires 306MB to store the executables instead of 252MB, or 54MB more. This is a very small fraction of a modern disk drive. Nonetheless, we believe that further reductions are possible. The current chunking algorithm does not take into account the internal structure of an ELF file. We believe that this structure can be exploited to improve chunk sharing between files, but we have not yet experimented with this.

4.3 Network Bandwidth

The third component of SLINKY is reducing the amount of network bandwidth required to transfer static executables. SLINKY accomplishes this by breaking the files into chunks, digesting the chunks, and transferring each unique chunk only once. Table 3 shows the results of our experiments with downloading Debian packages containing static vs. dynamic executables. We performed these experiments by writing a tool that takes a standard Debian package containing a dynamic executable (e.g. emacs), unpacks it, statically-links the executable using the `slink` tool, then repacks the package. We then downloaded the packages using `ckget` (Section 3.3). The *Dynamic* numbers in the table show the size of the dynamic packages (*Size*) and the number of bytes transferred when they are downloaded (*Xfer*). The packages were from the Debian “unstable” distribution.

Directory	SLINKY	Dynamic	Ratio
/bin	90.0	7.1	12.7
/sbin	123.1	5.5	22.2
/usr/bin	2945.8	219.6	13.4
/usr/sbin	244.5	25.1	9.7
/usr/X11R6/bin	396.4	37.3	10.6
All	3799.9	264.3	14.4

(a) Storage space required for statically- and dynamically-linked executables.

Directory	SLINKY	Dynamic	Ratio
/bin	7.0	7.0	1.0
/sbin	6.0	5.2	1.2
/usr/bin	251.6	208.9	1.2
/usr/sbin	26.3	25.0	1.0
/usr/X11R6/bin	43.0	36.2	1.2
All	305.9	251.9	1.2

(b) Storage space required for chunked executables.

Table 2: Storage space evaluation. All sizes are in MB.

The SLINKY numbers show that although the statically-linked packages are quite large, only a fraction of them need to be transferred when the packages are downloaded. The chunk database is initially empty on the receiving machine, representing a worst-case scenario for SLINKY as the dynamic packages already have their dependent libraries installed. The first package downloaded (**emacs**) transfers 11.9MB for the static package vs. 8.6MB for the dynamic, an increase of 39%. This is because the static package includes all of its dependent libraries. Note that the transfer size of 11.9MB for **emacs** is less than the 19.3MB size of the package; this is due to redundant chunks within the package. Subsequent static packages download fewer bytes than the package size because many chunks have already been downloaded. For **xutils** only 1.0MB of the 13.6MB package must be downloaded. Overall, the static packages required 34% more bytes to download than the comparable dynamic packages. This represents the worst-case situation for SLINKY as the chunk database was initially empty whereas the shared libraries on which the dynamic packages depend were already installed.

5 Related Work

SLINKY is unique in its use of digests to share data in memory, across the network, and on disk. Other systems have used digests or simple hashing to share data in some, but not all, of these areas. Waldspurger [22] describes a system called ESX Server that uses *content-based page sharing* to reduce the amount of memory required to run virtual machines on a virtual machine monitor. A background process scans the pages of physical memory and computes a simple hash of each page. Pages that have the same hash are compared, and identical pages are

Installation Workload	SLINKY		Dynamic	
	Size	Xfer	Size	Xfer
1> ckget emacs	19.3	11.9	8.6	8.6
2> ckget vim	5.4	4.2	3.6	3.6
3> ckget xmms	5.7	2.8	1.9	1.9
4> ckget xterm	2.4	0.9	0.5	0.5
5> ckget xutils	13.6	1.0	0.67	0.67
6> ckget xchat	0.5	0.5	0.5	0.5
Total	46.9	21.2	15.8	15.8

Table 3: Network bandwidth required to download SLINKY and dynamic Debian packages. The dynamic packages are the standard Debian packages; the SLINKY packages are chunked before compression. The SLINKY numbers include the packages' chunk files. All sizes are in MB.

shared copy-on-write. This allows the virtual machine monitor to share pages between virtual machines without any modification to the code the virtual machines run, or any understanding by the virtual machine monitor of the virtual machines it is running. Although both ESX Server and SLINKY share pages implicitly, the mechanisms for doing so are very different. ESX Server finds identical pages in a lazy fashion, searching the pool of existing pages for identical copies. This allows ESX Server to reduce the memory footprint without requiring digests as does SLINKY. However, hashing is not collision-free, so ESX server must compare pages when two hash to the same value. In contrast, SLINKY avoids creating duplicate copies of a page in the first place. Digests avoid having to compare pages with the same hash. SLINKY also shares only read-only pages, avoiding the need for a copy-on-write mechanism.

SLINKY's scheme for breaking a file into variable-

sized chunks using Rabin fingerprints is based on that of the Low-Bandwidth Network File System [11]. LBFS uses this scheme to reduce the amount of data required to transfer a file over the network, by sharing chunks of the file with other files already on the recipient (most notably previous versions of the same file). LBFS does not use digests to share pages in memory, nor does it use the chunking scheme to save space on disk. Instead, files are stored in a regular UNIX file system with an additional database that maps SHA-1 values to (file,offset,length) tuples to find the particular chunk.

The **rsync** [20] algorithm updates a file across a network. The recipient has an older version of the file, and computes the digests of fixed-size blocks. These digests are sent to the sender, who computes the digests of all overlapping fixed-size blocks. The sender then sends only those parts of the file that do not correspond to blocks already on the recipient.

Venti [14] uses SHA-1 hashes of fixed size blocks to store data in an archival storage system. Only one copy of each unique block need be stored, greatly reducing the storage requirements. Venti is block-based, and does not provide higher-level abstractions.

SFS-RO [6] is a read-only network file system that uses digests to provide secure file access. Entire files are named by their digest, and directories consist of (name, digest) pairs. File systems are named by the public key that corresponds to the private key used to sign the root digest. In this way files can be accessed securely from untrusted servers. SFS-RO differs from SLINKY in that it computes digests for entire files, and does not explicitly use the digests to reduce the amount of space required to store the data.

There are numerous tools to reduce the complexity of dynamic linking and shared libraries. Linux package systems such as **rpm** [17] and **dpkg** [5] were developed in part to deal with the dependencies between programs and libraries. Tools such as **apt** [2], **up2date** [21], and **yum** [23] download and install packages, and handle package dependencies by downloading and installing additional packages as necessary. In the Windows world, .NET provides facilities for avoiding DLL Hell [12]. The .NET framework provides an *assembly* abstraction that is similar to packages in Linux. Assemblies can either be private or shared, the former being the common case. Private assemblies allow applications to install the assemblies they need, independent of any assemblies already existing on the system. The net effect is for dynamically-linked executables to be shipped

with the dynamic libraries they need, and for each executable to have its own copy of its libraries. This obviously negates many of the purported advantages of shared libraries. Sharing and network transport is done at the level of assemblies, without any provisions for sharing content between assemblies.

Software configuration management tools such as **Vesta** [8] automatically discover dependencies between components and ensure consistency between builds. Such tools are indispensable to manage large software projects. However, they do not help with the DLL Hell problem that stems from inconsistencies arising on a user's machine as the result of installing binaries and libraries from a multitude of sources.

6 Conclusion

Static linking is the simplest way of combining separately compiled programs and libraries into an executable. A program and its libraries are simply merged into one file, and dependencies between them resolved. Distributing a statically linked program is also trivial — simply ship it to the user's machine where he can run it, regardless of what other programs and libraries are stored on his machine.

In this paper we have shown that the disadvantages associated with static linking (extra disk and memory space incurred by multiple programs linking against the same library, extra network transfer bandwidth being wasted during transport of the executables) can be largely eliminated. Our SLINKY system achieves this efficiency by use of digest-based sharing. Relative to dynamic linking, SLINKY has no measurable performance decrease, a comparable memory footprint, a storage space increase of 20%, and a network bandwidth increase of 34%. We are confident that additional tuning will improve these numbers. SLINKY thus makes it feasible to replace complicated dynamic linking with simple static linking.

Acknowledgments

This work was supported in part by the National Science Foundation under grant CCR-0073483.

References

- [1] O. Aoki and D. Sewell. Debian quick reference. <http://www.debian.org/doc/manuals/en/quick-reference.pdf>.
- [2] APT Howto. <http://www.debian.org/doc/manuals/apt-howto/index.en.html>.
- [3] F. Corbat and V. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS FJCC*, pages 185–196, 1965.
- [4] P. Davis. Ardour. <http://boudicca.tux.org/hypermil/ardour-dev/2002-Jun/0068.html>.
- [5] The dpkg package manager. <http://freshmeat.net/projects/dpkg>.
- [6] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *Computer Systems*, 20(1):1–24, 2002.
- [7] V. Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, July 2003.
- [8] A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta approach to software configuration management, 1999.
- [9] J. R. Levine. *Linkers and Loaders*. Morgan-Kaufman, 2000.
- [10] H. Lu. ELF: From the programmer’s perspective, 1995.
- [11] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.
- [12] S. Pratschner. Simplifying deployment and solving DLL Hell with the .NET framework. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dplywithnet.asp>, Nov. 2001.
- [13] prelink. <http://freshmeat.net/projects/prelink>.
- [14] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX Conference on File and Storage Technologies (FAST)*, Monterey, CA, 2002.
- [15] M. Rabin. Combinatorial algorithms on words. F12 of NATO ASI Series:279–288, 1985.
- [16] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6 (part 2)):1905+, 1978.
- [17] The RPM package manager. www.rpm.org.
- [18] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 2nd edition, 1996.
- [19] RFC 3174 - US secure hash algorithm 1 (SHA-1).
- [20] A. Tridgell and P. Macherras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, June 1996.
- [21] NRH-up2date. <http://www.nrh-up2date.org>.
- [22] C. A. Waldspurger. Memory resource management in VMware ESX server. In *5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, USA, 2002.
- [23] yum. <http://linux.duke.edu/projects/yum>.

CLOCK-Pro: An Effective Improvement of the CLOCK Replacement

Song Jiang

Feng Chen and Xiaodong Zhang

*Performance and Architecture Laboratory (PAL)
Los Alamos National Laboratory, CCS Division
Los Alamos, NM 87545, USA
sjiang@lanl.gov*

*Computer Science Department
College of William and Mary
Williamsburg, VA 23187, USA
{fchen, zhang}@cs.wm.edu*

Abstract

With the ever-growing performance gap between memory systems and disks, and rapidly improving CPU performance, virtual memory (VM) management becomes increasingly important for overall system performance. However, one of its critical components, the page replacement policy, is still dominated by CLOCK, a replacement policy developed almost 40 years ago. While pure LRU has an unaffordable cost in VM, CLOCK simulates the LRU replacement algorithm with a low cost acceptable in VM management. Over the last three decades, the inability of LRU as well as CLOCK to handle weak locality accesses has become increasingly serious, and an effective fix becomes increasingly desirable.

Inspired by our I/O buffer cache replacement algorithm, LIRS [13], we propose an improved CLOCK replacement policy, called CLOCK-Pro. By additionally keeping track of a limited number of replaced pages, CLOCK-Pro works in a similar fashion as CLOCK with a VM-affordable cost. Furthermore, it brings all the much-needed performance advantages from LIRS into CLOCK. Measurements from an implementation of CLOCK-Pro in Linux Kernel 2.4.21 show that the execution times of some commonly used programs can be reduced by up to 47%.

1 Introduction

1.1 Motivation

Memory management has been actively studied for decades. On one hand, to use installed memory effectively, much work has been done on memory allocation, recycling, and memory management in various programming languages. Many solutions and significant improvements have been seen in both theory and practice. On the other hand, aiming at reducing the cost of paging between memory and disks, researchers and practitioners in both academia and industry are working hard to improve the performance of page replacement, especially to avoid the worst performance cases. A significant advance in

this regard becomes increasingly demanding with the continuously growing gap between memory and disk access times, as well as rapidly improving CPU performance. Although increasing memory size can always reduce I/O pagings by giving a larger memory space to hold the working set, one cannot cache all the previously accessed data including file data in memory. Meanwhile, VM system designers should attempt to maximize the achievable performance under different application demands and system configurations. An effective replacement policy is critical in achieving the goal. Unfortunately, an approximation of LRU, the CLOCK replacement policy [5], which was developed almost 40 years ago, is still dominating nearly all the major operating systems including MVS, Unix, Linux and Windows [7]¹, even though it has apparent performance disadvantages inherited from LRU with certain commonly observed memory access behaviors.

We believe that there are two reasons responsible for the lack of significant improvements on VM page replacements. First, there is a very stringent cost requirement on the policy demanded by the VM management, which requires that the cost be associated with the number of page faults or a moderate constant. As we know, a page fault incurs a penalty worth of hundreds of thousands of CPU cycles. This allows a replacement policy to do its job without intrusively interfering with application executions. However, a policy with its cost proportional to the number of memory references would be prohibitively expensive, such as doing some bookkeeping on every memory access. This can cause the user program to generate a trap into the operating system on every memory instruction, and the CPU would consume much more cycles on page replacement than on user programs, even when there are no paging requests. From the cost perspective, even LRU, a well-recognized low-cost and simple replacement algorithm, is unaffordable, because it has to maintain the LRU ordering of pages for each page access. The second reason is that most proposed replacement algorithms attempting to improve LRU

¹This generally covers many CLOCK variants, including Mach-style active/inactive list, FIFO list facilitated with hardware reference bits. These CLOCK variants share similar performance problems plaguing LRU.

performance turn out to be too complicated to produce their approximations with their costs meeting the requirements of VM. This is because the weak cases for LRU mostly come from its minimal use of history access information, which motivates other researchers to take a different approach by adding more bookkeeping and access statistic analysis work to make their algorithms more intelligent in dealing with some access patterns unfriendly to LRU.

1.2 The Contributions of this Paper

The objective of our work is to provide a VM page replacement algorithm to take the place of CLOCK, which meets both the performance demand from application users and the low overhead requirement from system designers.

Inspired by the I/O buffer cache replacement algorithm, LIRS [13], we design an improved CLOCK replacement, called CLOCK-Pro. LIRS, originally invented to serve I/O buffer cache, has a cost unacceptable to VM management, even though it holds apparent performance advantages relative to LRU. We integrate the principle of LIRS and the way in which CLOCK works into CLOCK-Pro. By proposing CLOCK-Pro, we make several contributions: (1) CLOCK-Pro works in a similar fashion as CLOCK and its cost is easily affordable in VM management. (2) CLOCK-Pro brings all the much-needed performance advantages from LIRS into CLOCK. (3) Without any pre-determined parameters, CLOCK-Pro adapts to the changing access patterns to serve a broad spectrum of workloads. (4) Through extensive simulations on real-life I/O and VM traces, we have shown the significant page fault reductions of CLOCK-Pro over CLOCK as well as other representative VM replacement algorithms. (5) Measurement results from an implementation of CLOCK-Pro in a Linux kernel show that the execution times of some commonly used programs can be reduced by up to 47%.

2 Background

2.1 Limitations of LRU/CLOCK

LRU is designed on an assumption that a page would be re-accessed soon after it was accessed. It manages a data structure conventionally called LRU stack, in which the Most Recently Used (MRU) page is at the stack top and the Least Recently Used (LRU) page is at the stack bottom. The ordering of other in-between pages in the stack strictly follows their last access times. To maintain the stack, the LRU algorithm has to move an accessed page from its current position in the stack (if it is in the stack) to the stack top. The LRU page at the stack bottom is the one to be replaced if there is a page fault and no free spaces are available. In CLOCK, the memory spaces holding the pages can be regarded as a circular buffer and the replacement algorithm cycles through the pages in the

circular buffer, like the hand of a clock. Each page is associated with a bit, called reference bit, which is set by hardware whenever the page is accessed. When it is necessary to replace a page to service a page fault, the page pointed to by the hand is checked. If its reference bit is unset, the page is replaced. Otherwise, the algorithm resets its reference bit and keeps moving the hand to the next page. Research and experience have shown that CLOCK is a close approximation of LRU, and its performance characteristics are very similar to those of LRU. So all the performance disadvantages discussed below about LRU are also applied to CLOCK.

The LRU assumption is valid for a significant portion of workloads, and LRU works well for these workloads, which are called LRU-friendly workloads. The distance of a page in the LRU stack from the stack top to its current position is called **recency**, which is the number of other distinct pages accessed after the last reference to the page. Assuming an unlimitedly long LRU stack, the distance of a page in the stack away from the top when it is accessed is called its **reuse distance**, which is equivalent to the number of other distinct pages accessed between its last access and its current access. LRU-friendly workloads have two distinct characteristics: (1) There are much more references with small reuse distances than those with large reuse distances; (2) Most references have reuse distances smaller than the available memory size in terms of the number of pages. The locality exhibited in this type of workloads is regarded as strong, which ensures a high hit ratio and a steady increase of hit ratio with the increase of memory size.

However, there are indeed cases in which this assumption does not hold, where LRU performance could be unacceptably degraded. One example access pattern is memory scan, which consists of a sequence of one-time page accesses. These pages actually have infinitely large reuse distance and cause no hits. More seriously, in LRU, the scan could flush all the previously active pages out of memory.

As an example, in Linux the memory management for process-mapped program memory and file I/O buffer cache is unified, so the memory can be flexibly allocated between them according to their respective needs. The allocation balancing between program memory and buffer cache poses a big problem because of the unification. This problem is discussed in [22]. We know that there are a large amount of data in file systems, and the total number of accesses to the file cache could also be very large. However, the access frequency to each individual page of file data is usually low. In a burst of file accesses, most of the memory could serve as a file cache. Meanwhile, the process pages are evicted to make space for the actually infrequently accessed file pages, even though they are frequently accessed. An example scenario on this is that right after one extracts a large tarball, he/she could sense that the computer becomes slower because the previous active working set is replaced and has to be faulted in. To address this problem in a simple way, current Linux versions have to in-

roduce some “magic parameters” to enforce the buffer cache allocation to be in the range of 1% to 15% of memory size by default [22]. However, this approach does not fundamentally solve the problem, because the major reason causing the allocation unbalancing between process memory and buffer cache is the ineffectiveness of the replacement policy in dealing with infrequently accessed pages in buffer caches.

Another representative access pattern defeating LRU is loop, where a set of pages are accessed cyclically. Loop and loop-like access patterns dominate the memory access behaviors of many programs, particularly in scientific computation applications. If the pages involved in a loop cannot completely fit in the memory, there are repeated page faults and no hits at all. The most cited example for the loop problem is that even if one has a memory of 100 pages to hold 101 pages of data, the hit ratio would be ZERO for a looping over this data set [9, 24]!

2.2 LIRS and its Performance Advantages

A recent breakthrough in replacement algorithm designs, called LIRS (Low Inter-reference Recency Set) replacement [13], removes all the aforementioned LRU performance limitations while still maintaining a low cost close to LRU. It can not only fix the scan and loop problems, but also can accurately differentiate the pages based on their locality strengths quantified by reuse distance.

A key and unique approach in handling history access information in LIRS is that it uses reuse distance rather than recency in LRU for its replacement decision. In LIRS, a page with a large reuse distance will be replaced even if it has a small recency. For instance, when a one-time-used page is recently accessed in a memory scan, LIRS will replace it quickly because its reuse distance is infinite, even though its recency is very small. In contrast, LRU lacks the insights of LIRS: all accessed pages are indiscriminately cached until either of two cases happens to them: (1) they are re-accessed when they are in the stack, and (2) they are replaced at the bottom of the stack. LRU does not take account of which of the two cases has a higher probability. For infrequently accessed pages, which are highly possible to be replaced at the stack bottom without being re-accessed in the stack, holding them in memory (as well as in stack) certainly results in a waste of the memory resources. This explains the LRU misbehavior with the access patterns of weak locality.

3 Related Work

There have been a large number of new replacement algorithms proposed over the decades, especially in the last fifteen years. Almost all of them are proposed to target the performance problems of LRU. In general, there are three approaches taken in these algorithms. (1) Requiring applications

to explicitly provide future access hints, such as application-controlled file caching [3], and application-informed prefetching and caching [20]; (2) Explicitly detecting the access patterns failing LRU and adaptively switching to other effective replacements, such as SEQ [9], EELRU [24], and UBM [14]; (3) Tracing and utilizing deeper history access information such as FBR [21], LRFU [15], LRU-2 [18], 2Q [12], MQ [29], LIRS [13], and ARC [16]. More elaborate description and analysis on the algorithms can be found in [13]. The algorithms taking the first two approaches usually place too many constraints on the applications to be applicable in the VM management of a general-purpose OS. For example, SEQ is designed to work in VM management, and it only does its job when there is a page fault. However, its performance depends on an effective detection of long sequential address reference patterns, where LRU behaves poorly. Thus, SEQ loses generality because of the mechanism it uses. For instance, it is hard for SEQ to detect loop accesses over linked lists. Among the algorithms taking the third approach, FBR, LRU-2, LRFU and MQ are expensive compared with LRU. The performance of 2Q has been shown to be very sensitive to its parameters and could be much worse than LRU [13]. LIRS uses reuse distance, which has been used to characterize and to improve data access locality in programs (see e.g. [6]). LIRS and ARC are the two most promising candidate algorithms that have a potential leading to low-cost replacement policies applicable in VM, because they use data structure and operations similar to LRU and their cost is close to LRU.

ARC maintains two variably-sized lists holding history access information of referenced pages. Their combined size is two times of the number of pages in the memory. So ARC not only records the information of cached pages, but also keeps track of the same number of replaced pages. The first list contains pages that have been touched only once recently (*cold pages*) and the second list contains pages that have been touched at least twice recently (*hot pages*). The cache spaces allocated to the pages in these two lists are adaptively changed, depending on in which list the recent misses happen. More cache spaces will serve cold pages if there are more misses in the first list. Similarly, more cache spaces will serve hot pages if there are more misses in the second list. However, though ARC allocates memory to hot/cold pages adaptively according to the ratio of cold/hot page accesses and excludes tunable parameters, the locality of pages in the two lists, which are supposed to hold cold and hot pages respectively, can not directly and consistently be compared. So the hot pages in the second list could have a weaker locality in terms of reuse distance than the cold pages in the first list. For example, a page that is regularly accessed with a reuse distance a little bit more than the memory size can have no hits at all in ARC, while a page in the second list can stay in memory without any accesses, since it has been accepted into the list. This does not happen in LIRS, because any pages supposed to be hot or cold are placed in the same list and compared in a consistent fash-

ion. There is one pre-determined parameter in the LIRS algorithm on the amount of memory allocation for cold pages. In CLOCK-Pro, the parameter is removed and the allocation becomes fully adaptive to the current access patterns.

Compared with the research on the general replacement algorithms targeting LRU, the work specific to the VM replacements and targeting CLOCK is much less and is inadequate. While Second Chance (SC) [28], being the simplest variant of CLOCK algorithm, utilizes only one reference bit to indicate recency, other CLOCK variants introduce a finer distinction between page access history. In a generalized CLOCK version called GCLOCK [25, 17], a counter is associated with each page rather than a single bit. Its counter will be incremented if a page is hit. The cycling clock hand sweeps over the pages decrementing their counters until a page whose counter is zero is found for replacement. In Linux and FreeBSD, a similar mechanism called page aging is used. The counter is called *age* in Linux or *act_count* in FreeBSD. When scanning through memory for pages to replace, the page age is increased by a constant if its reference bit is set. Otherwise its age is decreased by a constant. One problem for this kind of design is that they cannot consistently improve LRU performance. The parameters for setting the maximum value of counters or adjusting ages are mostly empirically decided. Another problem is that they consume too many CPU cycles and adjust to changes of access patterns slowly, as evidenced in Linux kernel 2.0. Recently, an approximation version of ARC, called CAR [2], has been proposed, which has a cost close to CLOCK. Their simulation tests on the I/O traces indicate that CAR has a performance similar to ARC. The results of our experiments on I/O and VM traces show that CLOCK-Pro has a better performance than CAR.

In the design of VM replacements it is difficult to obtain much improvement in LRU due to its stringent cost constraint, yet this problem remains a demanding challenge in the OS development.

4 Description of CLOCK-Pro

4.1 Main Idea

CLOCK-Pro takes the same principle as that of LIRS – it uses the reuse distance (called IRR in LIRS) rather than recency in its replacement decision. When a page is accessed, the reuse distance is the period of time in terms of the number of other distinct pages accessed since its last access. Although there is a reuse distance between any two consecutive references to a page, only the most current distance is relevant in the replacement decision. We use the reuse distance of a page at the time of its access to categorize it either as a cold page if it has a large reuse distance, or as a hot page if it has a small reuse distance. Then we mark its status as being cold or hot. We place all the accessed pages, either hot or cold, into one

single list ² in the order of their accesses ³. In the list, the pages with small recencies are at the list head, and the pages with large recencies are at the list tail.

To give the cold pages a chance to compete with the hot pages and to ensure their cold/hot statuses accurately reflect their current access behavior, we grant a cold page a test period once it is accepted into the list. Then, if it is re-accessed during its test period, the cold page turns into a hot page. If the cold page passes the test period without a re-access, it will leave the list. Note that the cold page in its test period can be replaced out of memory, however, its page metadata remains in the list for the test purpose until the end of the test period or being re-accessed. When it is necessary to generate a free space, we replace a resident cold page.

The key question here is how to set the time of the test period. When a cold page is in the list and there is still at least one hot page after it (i.e., with a larger recency), it should turn into a hot page if it is accessed, because it has a new reuse distance smaller than the hot page(s) after it. Accordingly, the hot page with the largest recency should turn into a cold page. So the test period should be set as the largest recency of the hot pages. If we make sure that the hot page with the largest recency is always at the list tail, and all the cold pages that pass this hot page terminate their test periods, then the test period of a cold page is equal to the time before it passes the tail of the list. So all the non-resident cold pages can be removed from the list right after they reach the tail of the list. In practice, we could shorten the test period and limit the number of cold pages in the test period to reduce space cost. By implementing this testing mechanism, we make sure that “cold/hot” are defined based on relativity and by constant comparison in one clock, not on a fixed threshold that are used to separate the pages into two lists. This makes CLOCK-Pro distinctive from prior work including 2Q and CAR, which attempt to use a constant threshold to distinguish the two types of pages, and to treat them differently in their respective lists (2Q has two queues, and CAR has two clocks), which unfortunately causes these algorithms to share some of LRU’s performance weakness.

4.2 Data Structure

Let us first assume that the memory allocations for the hot and cold pages, m_h and m_c , respectively, are fixed, where $m_h + m_c$ is the total memory size m ($m = m_h + m_c$). The number of the hot pages is also m_h , so all the hot pages are always cached. If a hot page is going to be replaced, it must first change into a cold page. Apart from the hot pages, all the other accessed pages are categorized as cold pages. Among the cold pages, m_c pages are cached, another at most m non-resident

² Actually it is the metadata of a page that is placed in the list.

³ Actually we can only maintain an approximate access order, because we cannot update the list with a hit access in a VM replacement algorithm, thus losing the exact access orderings between page faults.

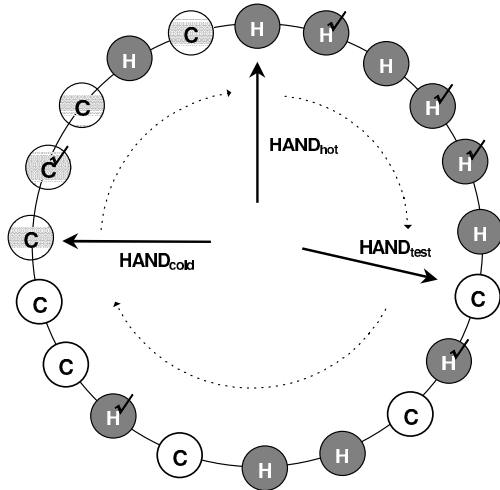


Figure 1: There are three types of pages in CLOCK-Pro, hot pages marked with ‘H’, cold pages marked with ‘C’ (shaded circles for resident cold pages, non-shaded circles for non-resident cold pages). Around the clock, there are three hands: $HAND_{hot}$ pointing to the list tail (i.e. the last hot page) and used to search for a hot page to turn into a cold page, $HAND_{cold}$ pointing to the last resident cold page and used to search for a cold page to replace, and $HAND_{test}$ pointing to the last cold page in the test period, terminating test periods of cold pages, and removing non-resident cold pages passing the test period out of the list. The ‘✓’ marks represent the reference bits of 1.

cold pages only have their history access information cached. So totally there are at most $2m$ metadata entries for keeping track of page access history in the list. As in CLOCK, all the page entries are organized as a circular linked list, shown in Figure 1. For each page, there is a cold/hot status associated with it. For each cold page, there is a flag indicating if the page is in the test period.

In CLOCK-Pro, there are three hands. The $HAND_{hot}$ points to the hot page with the largest recency. The position of this hand actually serves as a threshold of being a hot page. Any hot pages swept by the hand turn into cold ones. For the convenience of the presentation, we call the page pointed to by $HAND_{hot}$ as the tail of the list, and the page immediately after the tail page in the clockwise direction as the head of the list. $HAND_{cold}$ points to the last resident cold page (i.e., the furthest one to the list head). Because we always select this cold page for replacement, this is the position where we start to look for a victim page, equivalent to the hand in CLOCK. $HAND_{test}$ points to the last cold page in the test period. This hand is used to terminate the test period of cold pages. The non-resident cold pages swept over by this hand will leave the circular list. All the hands move in the clockwise direction.

4.3 Operations on Searching Victim Pages

Just as in CLOCK, there are no operations in CLOCK-Pro for page hits, only the reference bits of the accessed pages are

set by hardware. Before we see how a victim page is generated, let us examine how the three hands move around the clock, because the victim page is searched by coordinating the movements of the hands.

$HAND_{cold}$ is used to search for a resident cold page for replacement. If the reference bit of the cold page currently pointed to by $HAND_{cold}$ is unset, we replace the cold page for a free space. The replaced cold page will remain in the list as a non-resident cold page until it runs out of its test period, if it is in its test period. If not, we move it out of the clock. However, if its bit is set and it is in its test period, we turn the cold page into a hot page, and ask $HAND_{hot}$ for its actions, because an access during the test period indicates a competitively small reuse distance. If its bit is set but it is not in its test period, there are no status change as well as $HAND_{hot}$ actions. In both of the cases, its reference bit is reset, and we move it to the list head. The hand will keep moving until it encounters a cold page eligible for replacement, and stops at the next resident cold page.

As mentioned above, what triggers the movement of $HAND_{hot}$ is that a cold page is found to have been accessed in its test period and thus turns into a hot page, which maybe accordingly turns the hot page with the largest recency into a cold page. If the reference bit of the hot page pointed to by $HAND_{hot}$ is unset, we can simply change its status and then move the hand forward. However, if the bit is set, which indicates the page has been re-accessed, we spare this page, reset its reference bit and keep it as a hot page. This is because the actual access time of the hot page could be earlier than the cold page. Then we move the hand forward and do the same on the hot pages with their bits set until the hand encounters a hot page with a reference bit of zero. Then the hot page turns into a cold page. Note that moving $HAND_{hot}$ forward is equivalent to leaving the page it moves by at the list head. Whenever the hand encounters a cold page, it will terminate the page’s test period. The hand will also remove the cold page from the clock if it is non-resident (the most probable case). It actually does the work on the cold page on behalf of hand $HAND_{test}$. Finally the hand stops at a hot page.

We keep track of the number of non-resident cold pages. Once the number exceeds m , the memory size in the number of pages, we terminate the test period of the cold page pointed to by $HAND_{test}$. We also remove it from the clock if it is a non-resident page. Because the cold page has used up its test period without a re-access and has no chance to turn into a hot page with its next access. $HAND_{test}$ then moves forward and stops at the next cold page.

Now let us summarize how these hands coordinate their operations on the clock to resolve a page fault. When there is a page fault, the faulted page must be a cold page. We first run $HAND_{cold}$ for a free space. If the faulted cold page is not in the list, its reuse distance is highly likely to be larger than the recency of hot pages⁴. So the page is still categorized as a

⁴We cannot guarantee that it is a larger one because there are no opera-

cold page and is placed at the list head. The page also initiates its test period. If the number of cold pages is larger than the threshold ($m_c + m$), we run **HAND_{test}**. If the cold page is in the list⁵, the faulted page turns into a hot page and is placed at the head of the list. We run **HAND_{hot}** to turn a hot page with a large recency into a cold page.

4.4 Making CLOCK-Pro Adaptive

Until now, we have assumed that the memory allocations for the hot and cold pages are fixed. In LIRS, there is a pre-determined parameter, denoted as L_{hirs} , to measure the percentage of memory that are used by cold pages. As it is shown in [13], L_{hirs} actually affects how LIRS behaves differently from LRU. When L_{hirs} approaches 100%, LIRS's replacement behavior, as well as its hit ratios, are close to those of LRU. Although the evaluation of LIRS algorithm indicates that its performance is not sensitive to L_{hirs} variations within a large range between 1% and 30%, it also shows that the hit ratios of LIRS could be moderately lower than LRU for LRU-friendly workloads (i.e. with strong locality) and increasing L_{hirs} can eliminate the performance gap.

In CLOCK-Pro, resident cold pages are actually managed in the same way as in CLOCK. **HAND_{cold}** behaves the same as what the clock hand in CLOCK does: sweeping across the pages while sparing the page with a reference bit of 1 and replacing the page with a reference bit of 0. So increasing m_c , the size of the allocation for cold pages, makes CLOCK-Pro behave more like CLOCK.

Let us see the performance implication of changing memory allocation in CLOCK-Pro. To overcome the CLOCK performance disadvantages with weak access patterns such as scan and loop, a small m_c value means a quick eviction of cold pages just faulted in and the strong protection of hot pages from the interference of cold pages. However, for a strong locality access stream, almost all the accessed pages have relatively small reuse distance. But, some of the pages have to be categorized as cold pages. With a small m_c , a cold page would have to be replaced out of memory soon after its being loaded in. Due to its small reuse distance, the page is probably faulted in the memory again soon after its eviction and treated as a hot page because it is in its test period this time. This actually generates unnecessary misses for the pages with small reuse distances. Increasing m_c would allow these pages to be cached for a longer period of time and make it more possible for them to be re-accessed and to turn into hot pages without being replaced. Thus, they can save additional page faults.

For a given reuse distance of an accessed cold page, m_c decides the probability of a page being re-accessed before its

tions on hits in CLOCK-Pro and we limit the number of cold pages in the list. But our experiment results show this approximation minimally affects the performance of CLOCK-Pro.

⁵The cold page must be in its test period. Otherwise, it must have been removed from the list.

being replaced from the memory. For a cold page with its reuse distance larger than its test period, retaining the page in memory with a large m_c is a waste of buffer spaces. On the other hand, for a page with a small reuse distance, retaining the page in memory for a longer period of time with a large m_c would save an additional page fault. In the adaptive CLOCK-Pro, we allow m_c to dynamically adjust to the current reuse distance distribution. If a cold page is accessed during its test period, we increment m_c by 1. If a cold page passes its test period without a re-access, we decrement m_c by 1. Note the aforementioned cold pages include resident and non-resident cold pages. Once the m_c value is changed, the clock hands of CLOCK-Pro will realize the memory allocation by temporally adjusting the moving speeds of **HAND_{hot}** and **HAND_{cold}**.

With this adaptation, CLOCK-Pro could take both LRU advantages with strong locality and LIRS advantages with weak locality.

5 Performance Evaluation

We use both trace-driven simulations and prototype implementation to evaluate our CLOCK-Pro and to demonstrate its performance advantages. To allow us to extensively compare CLOCK-Pro with other algorithms aiming at improving LRU, including CLOCK, LIRS, CAR, and OPT, we built simulators running on the various types of representative workloads previously adopted for replacement algorithm studies. OPT is an optimal, but offline, unimplementable replacement algorithm [1]. We also implemented a CLOCK-Pro prototype in a Linux kernel to evaluate its performance as well as its overhead in a real system.

5.1 Trace-Driven Simulation Evaluation

Our simulation experiments are conducted in three steps with different kinds of workload traces. Because LIRS is originally proposed as an I/O buffer cache replacement algorithm, in the first step, we test the replacement algorithms on the I/O traces to see how well CLOCK-Pro can retain the LIRS performance merits, as well as its performance with typical I/O access patterns. In the second step, we test the algorithms on the VM traces of application program executions. Integrated VM management on file cache and program memory, as is implemented in Linux, is always desired. Because of the concern for mistreatment of file data and process pages as mentioned in Section 2.1, we test the algorithms on the aggregated VM and file I/O traces to see how these algorithms respond to the integration in the third step. We do not include the results of LRU in the presentation, because they are almost the same as those of CLOCK.

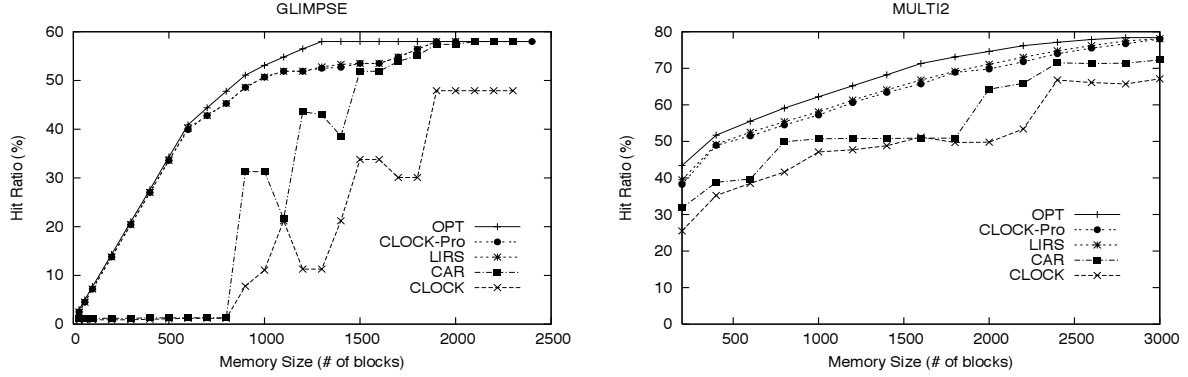


Figure 2: Hit ratios of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workloads *glimpse* and *multi2*.

5.1.1 Step 1: Simulation on I/O Buffer Caches

The file I/O traces used in this section are from [13] used for the LIRS evaluation. In their performance evaluation, the traces are categorized into four groups based on their access patterns, namely, loop, probabilistic, temporally-clustered and mixed patterns. Here we select one representative trace from each of the groups for the replacement evaluation, and briefly describe them here.

1. **glimpse** is a text information retrieval utility trace. The total size of text files used as input is roughly 50 MB. The trace is a member of the loop pattern group.
2. **cpp** is a GNU C compiler pre-processor trace. The total size of C source programs used as input is roughly 11 MB. The trace is a member of the probabilistic pattern group.
3. **sprite** is from the Sprite network file system, which contains requests to a file server from client workstations for a two-day period. The trace is a member of the temporally-clustered pattern group.
4. **multi2** is obtained by executing three workloads, cs, cpp, and postgres, together. The trace is a member of the mixed pattern group.

These are small-scale traces with clear access patterns. We use them to investigate the implications of various access patterns on the algorithms. The hit ratios of *glimpse* and *multi2* are shown in Figure 2. To help readers clearly see the hit ratio difference for *cpp* and *sprite*, we list their hit ratios in Tables 1 and 2, respectively. For LIRS, the memory allocation to HIR pages (L_{hirs}) is set as 1% of the memory size, the same value as it is used in [13]. There are several observations we can make on the results.

First, even though CLOCK-Pro does not responsively deal with hit accesses in order to meet the cost requirement of VM management, the hit ratios of CLOCK-Pro and LIRS are very close, which shows that CLOCK-Pro effectively retains the

blocks	OPT	CLOCK-Pro	LIRS	CAR	CLOCK
20	26.4	23.9	24.2	17.6	0.6
35	46.5	41.2	42.4	26.1	4.2
50	62.8	53.1	55.0	37.5	18.6
80	79.1	71.4	72.8	70.1	60.4
100	82.5	76.2	77.6	77.0	72.6
300	86.5	85.1	85.0	85.6	83.5
500	86.5	85.9	85.9	85.8	84.7
700	86.5	86.3	86.3	86.3	85.4
900	86.5	86.4	86.4	86.4	85.7

Table 1: Hit ratios (%) of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workload *cpp*.

blocks	OPT	CLOCK-Pro	LIRS	CAR	CLOCK
100	50.8	24.8	25.1	26.1	22.8
200	68.9	45.2	44.7	43.0	43.5
400	84.6	70.1	69.5	70.5	70.9
600	89.9	82.4	80.9	82.1	83.3
800	92.2	87.6	85.6	87.3	88.1
1000	93.2	89.7	87.6	89.6	90.4

Table 2: Hit ratios (%) of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workload *sprite*.

performance advantages of LIRS. For workloads *glimpse* and *multi2*, which contain many loop accesses, LIRS with a small L_{hirs} is most effective. The hit ratios of CLOCK-pro are a little lower than LIRS. However, for the LRU-friendly workload, *sprite*, which consists of strong locality accesses, the performance of LIRS could be lower than CLOCK (see Table 2). With its memory allocation adaptation, CLOCK-Pro improves the LIRS performance.

Figure 3 shows the percentage of the memory allocated to cold pages during the execution courses of *multi2* and *sprite* for a memory size of 600 pages. We can see that for *sprite*, the allocations for cold pages are much larger than 1% of the memory used in LIRS, and the allocation fluctuates over the

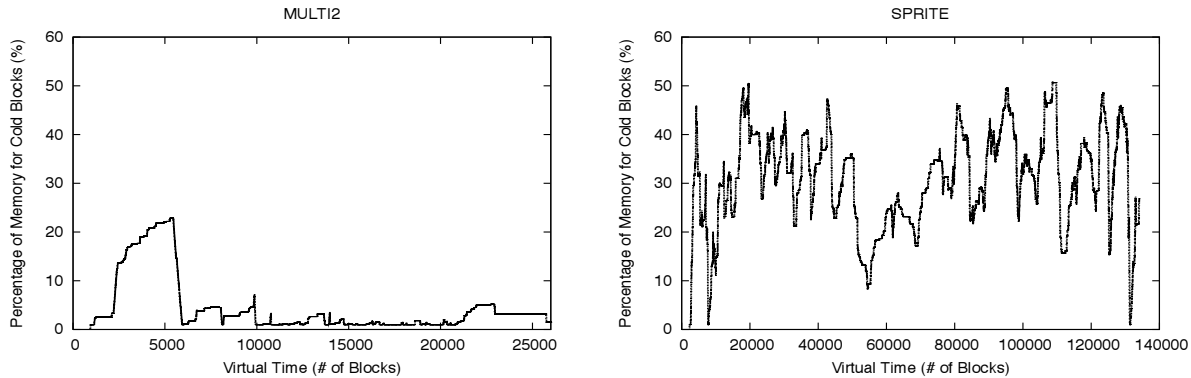


Figure 3: Adaptively changing the percentage of memory allocated to cold blocks in workloads *multi2* and *sprite*.

time adaptively to the changing access patterns. It sounds paradoxical that we need to increase the cold page allocation when there are many hot page accesses in the strong locality workload. Actually only the real cold pages with large reuse distances should be managed in a small cold allocation for their quick replacements. The so-called “cold” pages could actually be hot pages in strong locality workloads because the number of so-called “hot” pages are limited by their allocation. So quickly replacing these pseudo-cold pages should be avoided by increasing the cold page allocation. We can see that the cold page allocations for *multi2* are lower than *sprite*, which is consistent with the fact that *multi2* access patterns consist of many long loop accesses of weak locality.

Second, regarding the performance difference of the algorithms, CLOCK-Pro and LIRS have much higher hit ratios than CAR and CLOCK for *glimpse* and *multi2*, and are close to optimal. For strong locality accesses like *sprite*, there is little improvement either for CLOCK-Pro or CAR. This is why CLOCK is popular, considering its extremely simple implementation and low cost.

Third, even with a built-in memory allocation adaption mechanism, CAR cannot provide consistent improvements over CLOCK, especially for weak locality accesses, on which a fix is most needed for CLOCK. As we have analyzed, this is because CAR, as well as ARC, lacks a consistent locality strength comparison mechanism.

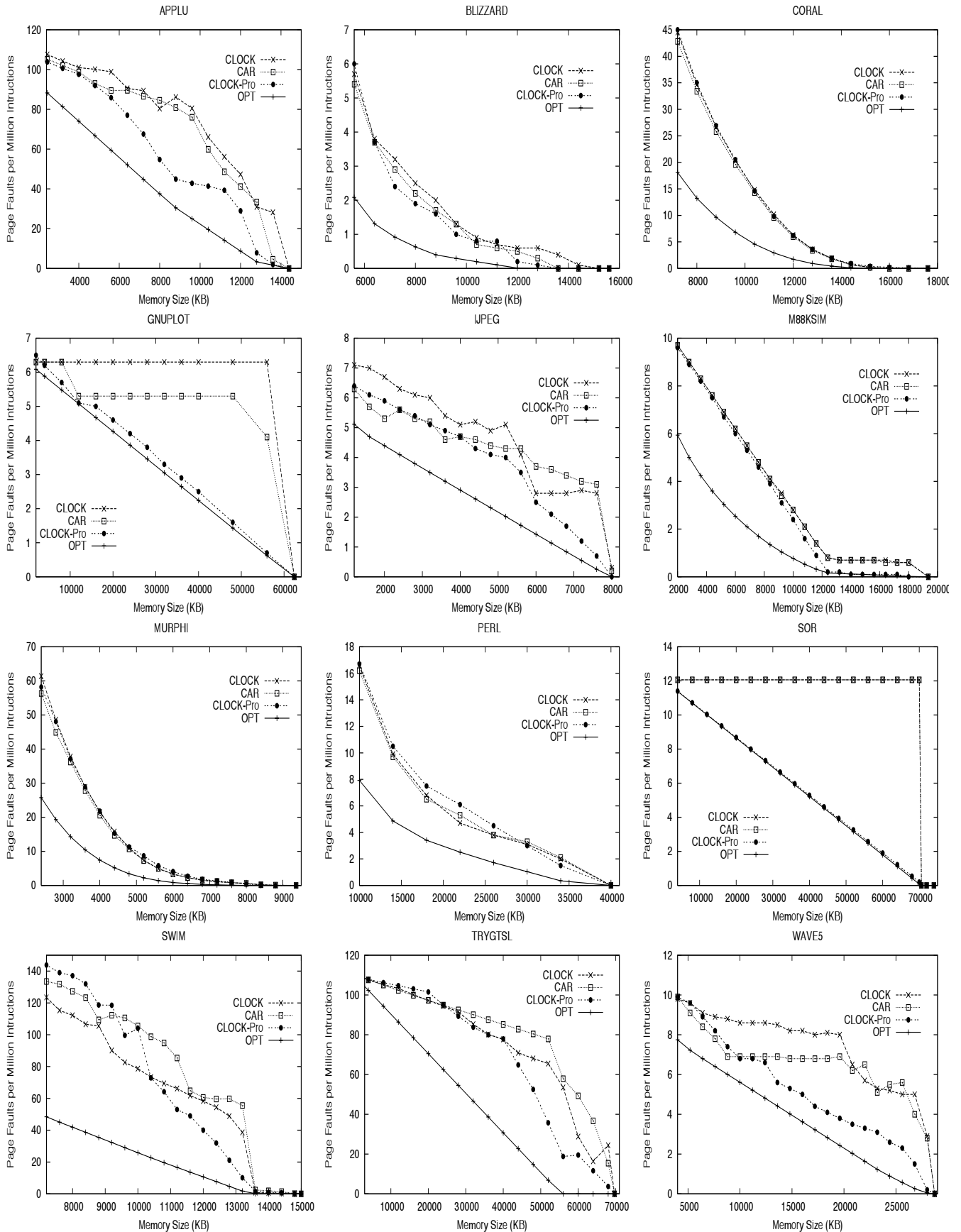
5.1.2 Step 2: Simulation on Memory for Program Executions

In this section, we use the traces of memory accesses of program executions to evaluate the performance of the algorithms. All the traces used here are also used in [10] and many of them are also used in [9, 24]. However, we do not include the performance results of SEQ and EELRU in this paper because of their generality or cost concerns for VM management. In some situations, EELRU needs to update its statistics for every single memory reference, having the same over-

head problem as LRU [24]. Interested readers are referred to the respective papers for detailed performance descriptions of SEQ and EELRU. By comparing the hit ratio curves presented in those papers with the curves provided in this paper about CLOCK-Pro (these results are comparable), readers will reach the conclusion that CLOCK-Pro provides better or equally good performance compared to SEQ and EELRU. Also because of overhead concern, we do not include the LRU and LIRS performance. Actually LRU has its hit ratio curves almost the same as those of CLOCK in our experiments.

Table 3 summarizes all the program traces used in this paper. The detailed program descriptions, space-time memory access graphs, and trace collection methodology, are described in [10, 9]. These traces cover a large range of various access patterns. After observing their memory access graphs drawn from the collected traces, the authors of paper [9] categorized programs *coral*, *m88ksim*, and *murphi* as having “no clearly visible patterns” with all accesses temporarily clustered together, categorized programs *blizzard*, *perl*, and *swim* as having “patterns at a small scale”, and categorized the rest of programs as having “clearly-exploitable, large-scale reference patterns”. If we examine the program access behaviors in terms of reuse distance, the programs in the first category are the strong locality workloads. Those in the second category are moderate locality workloads. And the remaining programs in the third category are weak locality workloads. Figure 4 shows the number of page faults per million of instructions executed for each of the programs, denoted as page fault ratio, as the memory increases up to its maximum memory demand. We exclude cold page faults which occur on their first time accesses. The algorithms considered here are CLOCK, CLOCK-Pro, CAR and OPT.

The simulation results clearly show that CLOCK-Pro significantly outperforms CLOCK for the programs with weak locality, including programs *applu*, *gunplot*, *jpeg*, *sor*, *trygtsl*, and *wave5*. For *gunplot* and *sor*, which have very large loop accesses, the page fault ratios of CLOCK-Pro are almost equal to those of OPT. The improvements of CAR over



Program	Description	Size (in Millions of Instructions)	Maximum Memory Demand (KB)
applu	Solve 5 coupled parabolic/elliptic PDE	1,068	14,524
blizzard	Binary rewriting tool for software DSM	2,122	15,632
coral	Deductive database evaluating query	4,327	20,284
gnuplot	PostScript graph generation	4,940	62,516
ijpeg	Image conversion into JPEG format	42,951	8,260
m88ksim	Microprocessor cycle-level simulator	10,020	19,352
murphi	Protocol verifier	1,019	9,380
perl	Interpreted scripting language	18,980	39,344
sor	Successive over-relaxation on a matrix	5,838	70,930
swim	Shallow water simulation	438	15,016
trytsl	Tridiagonal matrix calculation	377	69,688
wave5	plasma simulation	3,774	28,700

Table 3: A brief description of the programs used in Section 5.1.2.

CLOCK are far from being consistent and significant. In many cases, it performs worse than CLOCK. The poorest performance of CAR appears on traces *gunplot* and *sor* – it cannot correct the LRU problems with loop accesses and its page fault ratios are almost as high as those of CLOCK.

For programs with strong locality accesses, including *coral*, *m88ksim* and *murphi*, there is little room for other replacement algorithms to do a better job than CLOCK/LRU. Both CLOCK-Pro and ARC retain the LRU performance advantages for this type of programs, and CLOCK-Pro even does a little bit better than CLOCK.

For the programs with moderate locality accesses, including *blizzard*, *perl* and *swim*, the results are mixed. Though we see the improvements of CLOCK-Pro and CAR over CLOCK in the most cases, there does exist a case in *swim* with small memory sizes where CLOCK performs better than CLOCK-Pro and CAR. Though in most cases CLOCK-Pro performs better than CAR, for *perl* and *swim* with small memory sizes, CAR performs moderately better. After examining the traces, we found that the CLOCK-Pro performance variations are due to the working set shifting in the workloads. If a workload frequently shifts its working set, CLOCK-Pro has to actively adjust the composition of the hot page set to reflect current access patterns. When the memory size is small, the set of cold resident pages is small, which causes a cold/hot status exchange to be more possibly associated with an additional page fault. However, the existence of locality itself confines the extent of working set changes. Otherwise, no caching policy would fulfill its work. So we observed moderate performance degradations for CLOCK-Pro only with small memory sizes.

To summarize, we found that CLOCK-Pro can effectively remove the performance disadvantages of CLOCK in case of weak locality accesses, and CLOCK-Pro retains its performance advantages in case of strong locality accesses. It exhibits apparently more impressive performance than CAR, which was proposed with the same objectives as CLOCK-Pro.

5.1.3 Step 3: Simulation on Program Executions with Interference of File I/O

In an unified memory management system, file buffer cache and process memory are managed with a common replacement policy. As we have stated in Section 2.1, memory competition from a large number of file data accesses in a shared memory space could interfere with program execution. Because file data is far less frequently accessed than process VM, a process should be more competitive in preventing its memory from being taken away to be used as file cache buffer. However, recency-based replacement algorithms like CLOCK allow these file pages to replace process memory even if they are not frequently used, and to pollute the memory. To provide a preliminary study on the effect, we select an I/O trace (WebSearch1) from a popular search engine [26] and use its first 900 second accesses as a sample I/O accesses to co-occur with the process memory accesses in a shared memory space. This segment of I/O trace contains extremely weak locality – among the total 1.12 millions page accesses, there are 1.00 million unique pages accessed. We first scale the I/O trace onto the execution time of a program and then aggregate the I/O trace with the program VM trace in the order of their access times. We select a program with strong locality accesses, *m88ksim*, and a program with weak locality accesses, *sor*, for the study.

Tables 4 and 5 show the number of page faults per million of instructions (only the instructions for *m88ksim* or *sor* are counted) for *m88ksim* and *sor*, respectively, with various memory sizes. We are not interested in the performance of the I/O accesses. There would be few page hits even for a very large dedicated memory because there is little locality in their accesses.

From the simulation results shown in the tables, we observed that: (1) For the strong locality program, *m88ksim*, both CLOCK-Pro and CAR can effectively protect program execution from I/O access interference, while CLOCK is not able to reduce its page faults with increasingly large memory sizes. (2) For the weak locality program, *sor*, only CLOCK-

Memory(KB)	CLOCK-Pro	CLOCK-Pro w/IO	CAR	CAR w/IO	CLOCK	CLOCK w/IO
2000	9.6	9.94	9.7	10.1	9.7	11.23
3600	8.2	8.83	8.3	9.0	8.3	11.12
5200	6.7	7.63	6.9	7.8	6.9	11.02
6800	5.3	6.47	5.5	6.8	5.5	10.91
8400	3.9	5.22	4.1	5.8	4.1	10.81
10000	2.4	3.92	2.8	4.9	2.8	10.71
11600	0.9	2.37	1.4	4.2	1.4	10.61
13200	0.2	0.75	0.7	3.9	0.7	10.51
14800	0.1	0.52	0.7	3.6	0.7	10.41
16400	0.1	0.32	0.6	3.3	0.7	10.31
18000	0.0	0.22	0.6	3.1	0.6	10.22
19360	0.0	0.19	0.0	2.9	0.0	10.14

Table 4: The performance (number of page faults in one million of instructions) of algorithms CLOCK-Pro, CAR and CLOCK on program *m88ksim* with and without the interference of I/O file data accesses.

Memory(KB)	CLOCK-Pro	CLOCK-Pro w/IO	CAR	CAR w/IO	CLOCK	CLOCK w/IO
4000	11.4	11.9	12.1	12.2	12.1	12.2
12000	10.0	10.7	12.1	12.2	12.1	12.2
20000	8.7	9.6	12.1	12.2	12.1	12.2
28000	7.3	8.6	12.1	12.2	12.1	12.2
36000	5.9	7.5	12.1	12.2	12.1	12.2
44000	4.6	6.5	12.1	12.2	12.1	12.2
52000	3.2	5.4	12.1	12.2	12.1	12.2
60000	1.9	4.4	12.1	12.2	12.1	12.2
68000	0.5	3.4	12.1	12.2	12.1	12.2
70600	0.0	3.0	0.0	12.2	0.0	12.2
74000	0.0	2.6	0.0	12.2	0.0	12.2

Table 5: The performance (number of page faults in one million of instructions) of algorithms CLOCK-Pro, CAR and CLOCK on program *sor* with and without the interference of I/O file data accesses.

Pro can protect program execution from interference, though its page faults are moderately increased compared with its dedicated execution on the same size of memory. However, CAR and CLOCK cannot reduce their faults even when the memory size exceeds the program memory demand, and the number of faults on the dedicated executions has been zero.

We did not see a devastating influence on the program executions with the co-existence of the intensive file data accesses. This is because even the weak accesses of *m88ksim* are strong enough to stave off memory competition from file accesses with their page re-accesses, and actually there are almost no page reuses in the file accesses. However, if there are quiet periods during program active executions, such as waiting for user interactions, the program working set would be flushed by file accesses under recency-based replacement algorithms. Reuse distance based algorithms such as CLOCK-Pro will not have the problem, because file accesses have to generate small reuse distances to qualify the file data for a long-term memory stay, and to replace the program memory.

5.2 CLOCK-Pro Implementation and its Evaluation

The ultimate goal of a replacement algorithm is to reduce application execution times in a real system. In the process of translating the merits of an algorithm design to its practical performance advantages, many system elements could affect execution times, such as disk access scheduling, the gap between CPU and disk speeds, and the overhead of paging system itself. To evaluate the performance of CLOCK-Pro in a real system, we have implemented CLOCK-Pro in Linux kernel 2.4.21, which is a well documented recent version [11, 23].

5.2.1 Implementation and Evaluation Environment

We use a Gateway PC, which has its CPU of Intel P4 1.7GHz, its Western Digital WD400BB hard disk of 7200 RPM, and its memory of 256M. It is installed with RedHat 9. We are able to adjust the memory size available to the system and user programs by preventing certain portion of memory from being allocated.

In Kernel 2.4, process memory and file buffer are under an unified management. Memory pages are placed either in an

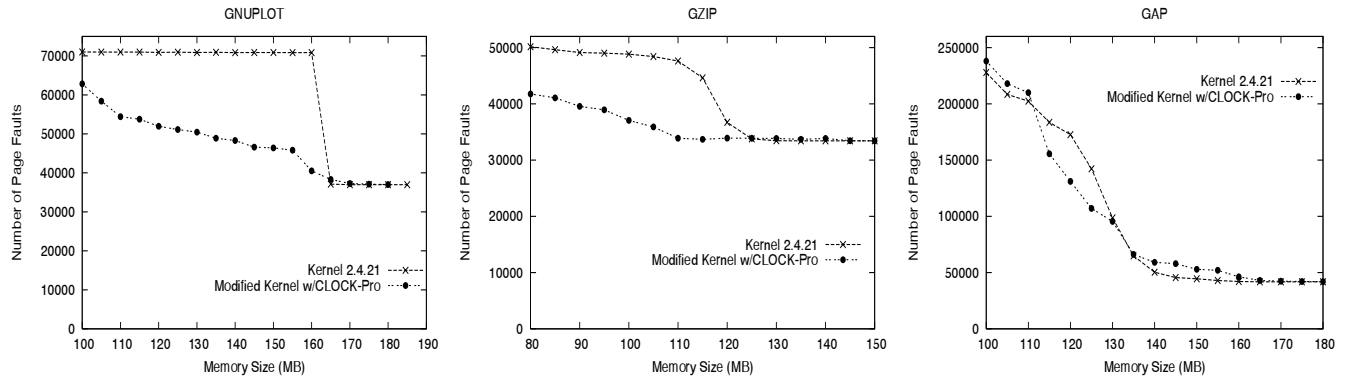


Figure 5: Measurements of the page faults of programs *gnuplot*, *gzip*, and *gap* on the original system and the system adopting CLOCK-Pro.

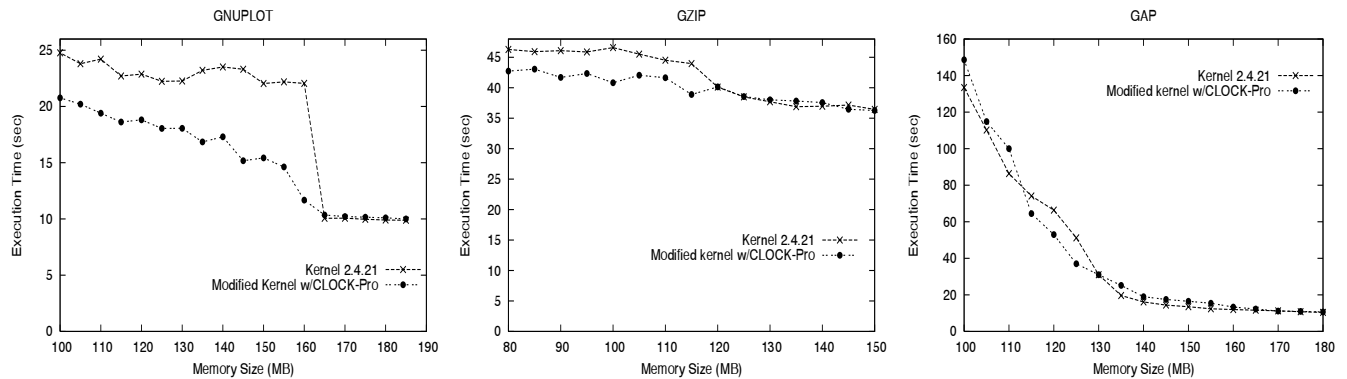


Figure 6: Measurements of the execution times of programs *gnuplot*, *gzip*, and *gap* on the original system and the system adopting CLOCK-Pro.

active list or in an inactive list. Each page is associated with a reference bit. When a page in the inactive list is detected being referenced, the page is promoted into the active list. Periodically the pages in the active list that are not recently accessed are removed to refill the inactive list. The kernel attempts to keep the ratio of the sizes of the active list and inactive list as 2:1. Each list is organized as a separate clock, where its pages are scanned for replacement or for movement between the lists. We notice that this kernel has adopted an idea similar to the 2Q replacement [12], by separating the pages into two lists to protect hot pages from being flushed by cold pages. However, a critical question still remains unanswered: how are the hot pages correctly identified from the cold pages?

This issue has been addressed in CLOCK-Pro, where we place all the pages in one single clock list, so that we can compare their hotness in a consistent way. To facilitate an efficient clock hand movement, each group of pages (with their statuses of hot, cold, and/or on test) are linked separately according to their orders in the clock list. The ratio of cold pages and hot pages is adaptively adjusted. CLOCK-Pro needs to keep track of a certain number of pages that have already been replaced from memory. We use their positions in the respective backup files to identify those pages, and maintain a hash table to efficiently retrieve their metadata when they are faulted in.

We ran SPEC CPU2000 programs and some commonly

used tools to test the performance of CLOCK-Pro as well as the original system. We observed consistent performance trends while running programs with weak, moderate, or strong locality on the original and modified systems. Here we present the representative results for three programs, each from one of the locality groups. Apart from *gnuplot*, a widely used interactive plotting program with its input data file of 16 MB, which we have used in our simulation experiments, the other two are from SPEC CPU2000 benchmark suite [27], namely, *gzip* and *gap*. *gzip* is a popular data compression program, showing a moderate locality. *gap* is a program implementing a language and library designed mostly for computing in groups, showing a strong locality. Both take the inputs from their respective training data sets.

5.2.2 Experimental Measurements

Figures 5 and 6 show the number of page faults and the execution times of programs *gnuplot*, *gzip*, and *gap* on the original system and the modified system adopting CLOCK-Pro. In the simulation-based evaluation, only page faults can be obtained. Here we also show the program execution times, which include page fault penalties and system paging overhead. It is noted that we include cold page faults in the statistics, because they contribute to the execution times. We see that the variations of the execution times with memory size generally

keep the same trends as those of page fault numbers, which shows that page fault is the major factor to affect system performance.

The measurements are consistent with the simulation results on the program traces shown in Section 5.1.2. For the weak locality program *gnuplot*, CLOCK-Pro significantly improves its performance by reducing both its page fault numbers and its execution times. The largest performance improvement comes at around 160MB, the available memory size approaching the memory demand, where the time for CLOCK-Pro (11.7 sec) is reduced by 47% when compared with the time for the original system (22.1 sec). There are some fluctuations in the execution time curves. This is caused by the block layout on the disk. A page faulted in from a disk position sequential to the previous access position has a much smaller access time than that retrieved from a random position. So the penalty varies from one page fault to another. For programs *gzip* and *gap* with a moderate or strong locality, CLOCK-Pro provides a performance as good as the original system.

Currently this is only a prototype implementation of CLOCK-Pro, in which we have attempted to minimize the changes in the existing data structures and functions, and make the most of the existing infrastructure. Sometimes this means a compromise in the CLOCK-Pro performance. For example, the hardware MMU automatically sets the reference bits on the *pte* (Page Table Entry) entries of a process page table to indicate the references to the corresponding pages. In kernel 2.4, the paging system works on the active or inactive lists, whose entries are called *page descriptors*. Each *descriptor* is associated with one physical page and one or more (if the page is shared) *pte* entries in the process page tables. Each *descriptor* contains a *reference flag*, whose value is transferred from its associated *pte* when the corresponding process table is scanned. So there is an additional delay for the reference bits (flags) to be seen by the paging system. In kernel 2.4, there is no infrastructure supporting the retrieval of *pte* through the *descriptor*. So we have to accept this delay in the implementation. However, this tolerance is especially detrimental to CLOCK-Pro because it relies on a fine-grained access timing distinction to realize its advantages. We believe that further refinement and tuning of the implementation will exploit more performance potential of CLOCK-Pro.

5.2.3 The Overhead of CLOCK-Pro

Because we almost keep the paging infrastructure of the original system intact except replacing the active/inactive lists with an unified clock list and introducing a hash table, the additional overhead from CLOCK-Pro is limited to the clock list and hash table operations.

We measure the average number of entries the clock hands sweep over per page fault on the lists for the two systems. Table 6 shows a sample of the measurements. The results show that CLOCK-Pro has a number of hand movements compa-

Memory (MB)	110	140	170
Kernel 2.4.21	12.4	14.2	6.9
CLOCK-Pro	16.2	20.6	18.5

Table 6: Average number of entries the clock hands sweep over per page fault in the original kernel and CLOCK-Pro with different memory sizes for program *gnuplot*.

table to the original system except for large memory sizes, where the original system significantly lowers its movement number while CLOCK-Pro does not. In CLOCK-Pro, for every referenced cold page seen by the moving **HAND_{cold}**, there is at least one **HAND_{hot}** movement to exchange the page statuses. For a specific program with a stable locality, there are fewer cold pages with a smaller memory, as well as less possibility for a cold page to be re-referenced before **HAND_{cold}** moves to it. So **HAND_{cold}** can take a small number of movements to reach a qualified replacement page, and the number of additional **HAND_{hot}** movements per page fault is also small. When the memory size is close to the program memory demand, the original system can take less hand movements during its search on its inactive list, due to the increasing chance of finding an unreferenced page. However, **HAND_{cold}** would encounter more referenced cold pages, which causes additional **HAND_{hot}** movements. We believe that this is not a performance concern, because one page fault penalty is equivalent to the time of tens of thousands of hand movements. We also measured the bucket size of the hash table, which is only 4-5 on average. So we conclude that the additional overhead is negligible compared with the original replacement implementation.

6 Conclusions

In this paper, we propose a new VM replacement policy, CLOCK-Pro, which is intended to take the place of CLOCK currently dominating various operating systems. We believe it is a promising replacement policy in the modern OS designs and implementations for the following reasons. (1) It has a low cost that can be easily accepted by current systems. Though it could move up to three pointers (hands) during one victim page search, the total number of the hand movements is comparable to that of CLOCK. Keeping track of the replaced pages in CLOCK-Pro doubles the size of the linked list used in CLOCK. However, considering the marginal memory consumption of the list in CLOCK, the additional cost is negligible. (2) CLOCK-pro provides a systematic solution to address the CLOCK problems. It is not just a quick and experience-based fix to CLOCK in a specific situation, but is designed based on a more accurate locality definition – reuse distance and addresses the source of the LRU problem. (3) It is fully adaptive to strong or weak access patterns without any pre-determined parameters. (4) Extensive simulation experiments

and a prototype implementation show its significant and consistent performance improvements.

Acknowledgments

We are grateful to our shepherd Yuanyuan Zhou and the anonymous reviewers who helped further improve the quality of this paper. We thank our colleague Bill Bynum for reading the paper and his comments. The research is partially supported by the National Science Foundation under grants CNS-0098055, CCF-0129883, and CNS-0405909.

References

- [1] L. A. Belady "A Study of Replacement Algorithms for Virtual Storage", *IBM System Journal*, 1966.
- [2] S. Bansal and D. Modha, "CAR: Clock with Adaptive Replacement", *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies*, March, 2004.
- [3] P. Cao, E. W. Felten and K. Li, "Application-Controlled File Caching Policies", *Proceedings of the USENIX Summer 1994 Technical Conference*, 1994.
- [4] J. Choi, S. Noh, S. Min, Y. Cho, "An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme", *Proceedings of the 1999 USENIX Annual Technical Conference*, 1999, pp. 239-252.
- [5] F. J. Corbato, "A Paging Experiment with the Multics System", *MIT Project MAC Report MAC-M-384*, May, 1968.
- [6] C. Ding and Y. Zhong, "Predicting Whole-Program Locality through Reuse-Distance Analysis", *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June, 2003.
- [7] M. B. Friedman, "Windows NT Page Replacement Policies", *Proceedings of 25th International Computer Measurement Group Conference*, Dec, 1999, pp. 234-244.
- [8] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management", *ACM Transaction on Database Systems*, Dec, 1984, pp. 560-595.
- [9] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior", *Proceedings of 1997 ACM SIGMETRICS Conference*, May 1997, pp. 115-126.
- [10] G. Glass, "Adaptive Page Replacement". Master's Thesis, University of Wisconsin, 1997.
- [11] M. Gorman, "Understanding the Linux Virtual Memory Manager", *Prentice Hall*, April, 2004.
- [12] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm", *Proceedings of the 20th International Conference on VLDB*, 1994, pp. 439-450.
- [13] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance", *In Proceeding of 2002 ACM SIGMETRICS*, June 2002, pp. 31-42.
- [14] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim "A Low-Overhead, High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References", *4th Symposium on Operating System Design & Implementation*, October 2000.
- [15] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho and C. Kim, "On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies", *Proceeding of 1999 ACM SIGMETRICS Conference*, May 1999.
- [16] N. Megiddo and D. Modha, "ARC: a Self-tuning, Low Overhead Replacement Cache", *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies*, March, 2003.
- [17] V. F. Nicola, A. Dan, and D. M. Dias, "Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing", *Proceeding of 1992 ACM SIGMETRICS Conference*, June 1992, pp. 35-46.
- [18] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering", *Proceedings of the 1993 ACM SIGMOD Conference*, 1993, pp. 297-306.
- [19] V. Phalke and B. Gopinath, "An Inter-Reference gap Model for Temporal Locality in Program Behavior", *Proceeding of 1995 ACM SIGMETRICS Conference*, May 1995.
- [20] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky and J. Zelenka, "Informed Prefetching and Caching", *Proceedings of the 15th Symposium on Operating System Principles*, 1995, pp. 1-16.
- [21] J. T. Robinson and N. V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement", *Proceeding of 1990 ACM SIGMETRICS Conference*, 1990.
- [22] R. van Riel, "Towards an O(1) VM: Making Linux Virtual Memory Management Scale Towards Large Amounts of Physical Memory", *Proceedings of the Linux Symposium*, July 2003.
- [23] R. van Riel, "Page Replacement in Linux 2.4 Memory Management", *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June, 2001.
- [24] Y. Smaragdakis, S. Kaplan, and P. Wilson, "The EELRU adaptive replacement algorithm", *Performance Evaluation (Elsevier)*, Vol. 53, No. 2, July 2003.
- [25] A. J. Smith, "Sequentiality and Prefetching in Database Systems", *ACM Trans. on Database Systems*, Vol. 3, No. 3, 1978, pp. 223-247.
- [26] Storage Performance Council, <http://www.storageperformance.org>
- [27] Standard Performance Evaluation Corporation, SPEC CPU2000 V1.2, <http://www.spec.org/cpu2000/>
- [28] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems, Design and Implementation*, Prentice Hall, 1997.
- [29] Y. Zhou, Z. Chen and K. Li. "Second-Level Buffer Cache Management", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 7, July, 2004.

Group Ratio Round-Robin: $O(1)$ Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems

Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein*, and Haoqiang Zheng
Department of Computer Science
Columbia University
Email: {bc2008, wc164, nieh, cliff, hzheng}@cs.columbia.edu

Abstract

We present Group Ratio Round-Robin (GR^3), the first proportional share scheduler that combines accurate proportional fairness scheduling behavior with $O(1)$ scheduling overhead on both uniprocessor and multiprocessor systems. GR^3 uses a simple grouping strategy to organize clients into groups of similar processor allocations which can be more easily scheduled. Using this strategy, GR^3 combines the benefits of low overhead round-robin execution with a novel ratio-based scheduling algorithm. GR^3 introduces a novel frontlog mechanism and weight readjustment algorithm to operate effectively on multiprocessors. GR^3 provides fairness within a constant factor of the ideal generalized processor sharing model for client weights with a fixed upper bound and preserves its fairness properties on multiprocessor systems. We have implemented GR^3 in Linux and measured its performance. Our experimental results show that GR^3 provides much lower scheduling overhead and much better scheduling accuracy than other schedulers commonly used in research and practice.

1 Introduction

Proportional share resource management provides a flexible and useful abstraction for multiplexing processor resources among a set of clients. Proportional share scheduling has a clear colloquial meaning: given a set of clients with associated weights, a proportional share scheduler should allocate resources to each client in proportion to its respective weight. However, developing processor scheduling mechanisms that combine good proportional fairness scheduling behavior with low scheduling overhead has been difficult to achieve in practice. For many proportional share scheduling mechanisms, the time to select a client for execution grows with the number of clients. For server systems which may service large numbers of clients, the scheduling overhead of algorithms whose complexity grows linearly with the number of clients can waste more than 20 percent of system resources [3] for large numbers of clients. Furthermore, little

work has been done to provide proportional share scheduling on multiprocessor systems, which are increasingly common especially in small-scale configurations with two or four processors. Over the years, a number of scheduling mechanisms have been proposed, and much progress has been made. However, previous mechanisms have either superconstant overhead or less-than-ideal fairness properties.

We introduce Group Ratio Round-Robin (GR^3), the first proportional share scheduler that provides constant fairness bounds on proportional sharing accuracy with $O(1)$ scheduling overhead for both uniprocessor and small-scale multiprocessor systems. In designing GR^3 , we observed that accurate, low-overhead proportional sharing is easy to achieve when scheduling a set of clients with equal processor allocations, but is harder to do when clients require very different allocations. Based on this observation, GR^3 uses a simple client grouping strategy to organize clients into groups of similar processor allocations which can be more easily scheduled. Using this grouping strategy, GR^3 combines the benefits of low overhead round-robin execution with a novel ratio-based scheduling algorithm.

GR^3 uses the same basic uniprocessor scheduling algorithm for multiprocessor scheduling by introducing the notion of a frontlog. On a multiprocessor system, a client may not be able to be scheduled to run on a processor because it is currently running on another processor. To preserve its fairness properties, GR^3 keeps track of a frontlog per client to indicate when the client was already running but could have been scheduled to run on another processor. It then assigns the client a time quantum that is added to its allocation on the processor it is running on. The frontlog ensures that a client receives its proportional share allocation while also taking advantage of any cache affinity by continuing to run the client on the same processor.

GR^3 provides a simple weight readjustment algorithm that takes advantage of its grouping strategy. On a multiprocessor system, proportional sharing is not feasible for some client weight assignments, such as having one client with weight 1 and another with weight 2 on a

*also in Department of IEOR

two-processor system. By organizing clients with similar weights into groups, GR^3 adjusts for infeasible weight assignments without the need to order clients, resulting in lower scheduling complexity than previous approaches [7].

We have analyzed GR^3 and show that with only $O(1)$ overhead, GR^3 provides fairness within $O(g^2)$ of the ideal Generalized Processor Sharing (GPS) model [16], where g , the number of groups, grows at worst logarithmically with the largest client weight. Since g is in practice a small constant, GR^3 effectively provides constant fairness bounds with only $O(1)$ overhead. Moreover, we show that GR^3 uniquely preserves its worst-case time complexity and fairness properties for multiprocessor systems.

We have implemented a prototype GR^3 processor scheduler in Linux, and compared it against uniprocessor and multiprocessor schedulers commonly used in practice and research, including the standard Linux scheduler [2], Weighted Fair Queueing (WFQ) [11], Virtual-Time Round-Robin (VTRR) [17], and Smoothed Round-Robin (SRR) [9]. We have conducted both simulation studies and kernel measurements on micro-benchmarks and real applications. Our results show that GR^3 can provide more than an order of magnitude better proportional sharing accuracy than these other schedulers, in some cases with more than an order of magnitude less overhead. These results demonstrate that GR^3 can in practice deliver better proportional share control with lower scheduling overhead than these other approaches. Furthermore, GR^3 is simple to implement and easy to incorporate into existing scheduling frameworks in commodity operating systems.

This paper presents the design, analysis, and evaluation of GR^3 . Section 2 describes the uniprocessor scheduling algorithm. Section 3 describes extensions for multiprocessor scheduling, which we refer to as GR^3MP . Section 4 analyzes the fairness and complexity of GR^3 . Section 5 presents experimental results. Section 6 discusses related work. Finally, we present some concluding remarks and directions for future work.

2 GR^3 Uniprocessor Scheduling

Uniprocessor scheduling, the process of scheduling a time-multiplexed resource among a set of clients, has two basic steps: 1) order the clients in a queue, 2) run the first client in the queue for its *time quantum*, which is the maximum time interval the client is allowed to run before another scheduling decision is made. We refer to the units of time quanta as time units (tu) rather than an absolute time measure such as seconds. A scheduler can therefore achieve proportional sharing in one of two ways. One way, often called fair queueing [11, 18, 28, 13, 24, 10] is to adjust the frequency that a client is selected to run by adjusting the position of the client in the queue so that it ends up at the front of the queue more or less often. However, adjusting the client's position in the queue typically requires sorting clients based

on some metric of fairness, and has a time complexity that grows with the number of clients. The other way is to adjust the size of a client's time quantum so that it runs longer for a given allocation, as is done in weighted round-robin (WRR). This is fast, providing constant time complexity scheduling overhead. However, allowing a client to monopolize the resource for a long period of time results in extended periods of unfairness to other clients which receive no service during those times. The unfairness is worse with skewed weight distributions.

GR^3 is a proportional share scheduler that matches with $O(1)$ time complexity of round-robin scheduling but provides much better proportional fairness guarantees in practice. At a high-level, the GR^3 scheduling algorithm can be briefly described in three parts:

1. **Client grouping strategy:** Clients are separated into groups of clients with similar weight values. The group of order k is assigned all clients with weights between 2^k to $2^{k+1} - 1$, where $k \geq 0$.
2. **Inter-group scheduling:** Groups are ordered in a list from largest to smallest group weight, where the group weight of a group is the sum of the weights of all clients in the group. Groups are selected in a round-robin manner based on the ratio of their group weights. If a group has already been selected more than its proportional share of the time, move on to the next group in the list. Otherwise, skip the remaining groups in the group list and start selecting groups from the beginning of the group list again. Since the groups with larger weights are placed first in the list, this allows them to get more service than the lower-weight groups at the end of the list.
3. **Intra-group scheduling:** From the selected group, a client is selected to run in a round-robin manner that accounts for its weight and previous execution history.

Using this client grouping strategy, GR^3 separates scheduling in a way that reduces the need to schedule entities with skewed weight distributions. The client grouping strategy limits the number of groups that need to be scheduled since the number of groups grows at worst logarithmically with the largest client weight. Even a very large 32-bit client weight would limit the number of groups to no more than 32. The client grouping strategy also ensures that all clients within a group have weight within a factor of two. As a result, the intra-group scheduler never needs to schedule clients with skewed weight distributions. GR^3 groups are simple lists that do not need to be balanced; they do not require any use of more complex balanced tree structures.

2.1 GR^3 Definitions

We now define the state GR^3 associates with each client and group, and describe in detail how GR^3 uses

C_j	Client j . (also called 'task' j)
ϕ_C	The weight assigned to client C .
ϕ_j	Shorthand notation for ϕ_{C_j} .
D_C	The deficit of C .
N	The number of runnable clients.
g	The number of groups.
G_i	i 'th group in the list ordered by weight.
$ G $	The number of clients in group G .
$G(C)$	The group to which C belongs.
Φ_G	The group weight of G : $\sum_{C \in G} \phi_C$.
Φ_i	Shorthand notation for Φ_{G_i} .
σ_G	The order of group G .
ϕ_{\min}^G	Lower bound for client weights in G : 2^{σ_G} .
w_C	The work of client C .
w_j	Shorthand notation for w_{C_j} .
W_G	The group work of group G .
W_i	Shorthand notation for W_{G_i} .
Φ_T	Total weight: $\sum_{j=1}^N \phi_j = \sum_{i=1}^g \Phi_i$.
W_T	Total work: $\sum_{j=1}^N w_j = \sum_{i=1}^g W_i$.
e_C	Service error of client C : $w_C - W_T \frac{\phi_C}{\Phi_T}$.
E_G	Group service error of G : $W_G - W_T \frac{\Phi_G}{\Phi_T}$.
$e_{C,G}$	Group-relative service error of client C with respect to group G : $w_C - W_G \frac{\phi_C}{\Phi_G}$.

Table 1: GR^3 terminology

that state to schedule clients. Table 1 lists terminology we use. For each client, GR^3 maintains the following three values: weight, deficit, and run state. Each client receives a resource allocation that is directly proportional to its *weight*. A client's *deficit* tracks the number of remaining time quanta the client has not received from previous allocations. A client's *run state* indicates whether or not it can be executed. A client is *runnable* if it can be executed.

For each group, GR^3 maintains the following four values: group weight, group order, group work, and current client. The *group weight* is the sum of the corresponding weights of the clients in the group run queue. A group with *group order* k contains the clients with weights between 2^k to $2^{k+1} - 1$. The *group work* is the total execution time clients in the group have received. The *current client* is the most recently scheduled client in the group's run queue.

GR^3 also maintains the following scheduler state: time quantum, group list, total weight, and current group. The *group list* is a sorted list of all groups containing runnable clients ordered from largest to smallest group weight, with ties broken by group order. The *total weight* is the sum of the weights of all runnable clients. The *current group* is the most recently selected group in the group list.

2.2 Basic GR^3 Algorithm

We initially only consider runnable clients in our discussion of the basic GR^3 scheduling algorithm. We dis-

cuss dynamic changes in a client's run state in Section 2.3. We first focus on the GR^3 intergroup scheduling algorithm, then discuss the GR^3 intragroup scheduling algorithm.

The GR^3 **intergroup scheduling** algorithm uses the ratio of the group weights of successive groups to determine which group to select. The next group to schedule is selected using only the state of successive groups in the group list. Given a group G_i whose weight is x times larger than the group weight of the next group G_{i+1} in the group list, GR^3 will select group G_i x times for every time that it selects G_{i+1} in the group list to provide proportional share allocation among groups.

To implement the algorithm, GR^3 maintains the total work done by group G_i in a variable W_i . An index i to tracks the current group and is initialized to 1. The scheduling algorithm then executes the following simple routine:

```

INTERGROUP-SCHEDULE()
1   $C \leftarrow \text{INTRAGROUP-SCHEDULE}(G_i)$ 
2   $W_i \leftarrow W_i + 1$ 
3  if  $i < g$  and  $\frac{W_i+1}{W_{i+1}+1} > \frac{\Phi_i}{\Phi_{i+1}}$  (1)
4    then  $i \leftarrow i + 1$ 
5    else  $i \leftarrow 1$ 
6  return  $C$ 

```

Let us negate (1) under the form:

$$\frac{W_i + 1}{\Phi_i} \leq \frac{W_{i+1} + 1}{\Phi_{i+1}} \quad (2)$$

We will call this relation the *well-ordering condition* of two consecutive groups. GR^3 works to maintain this condition true at all times. The intuition behind (2) is that we would like the ratio of the work of G_i and G_{i+1} to match the ratio of their respective group weights after GR^3 has finished selecting both groups. Recall, $\Phi_i \geq \Phi_{i+1}$. Each time a client from G_{i+1} is run, GR^3 would like to have run Φ_i/Φ_{i+1} worth of clients from G_i . (1) says that GR^3 should not run a client from G_i and increment G_i 's group work if it will make it impossible for G_{i+1} to catch up to its proportional share allocation by running one of its clients once.

To illustrate how intergroup scheduling works, Figure 1 shows an example with three clients C_1 , C_2 , and C_3 , which have weights of 5, 2, and 1, respectively. The GR^3 grouping strategy would place each C_i in group G_i , ordering the groups by weight: G_1 , G_2 , and G_3 have orders 2, 1 and 0 and weights of 5, 2, and 1 respectively. In this example, each group has only one client so there is no intragroup scheduling. GR^3 would start by selecting group G_1 , running client C_1 , and incrementing W_1 . Based on (1), $\frac{W_1+1}{W_2+1} = 2 < \frac{\Phi_1}{\Phi_2} = 2.5$, so GR^3 would select G_1 again and run client C_1 . After running C_1 , G_1 's work would be 2 so that the inequality in (1) would hold and GR^3 would then move on to the next group G_2 and run client C_2 . Based on (1), $\frac{W_2+1}{W_3+1} = 2 \leq \frac{\Phi_2}{\Phi_3} = 2$, so GR^3 would reset

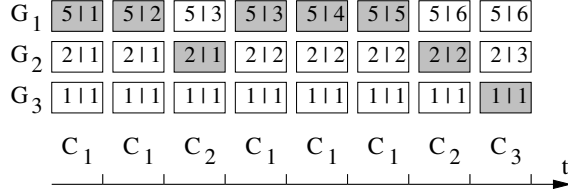


Figure 1: GR^3 intergroup scheduling. At each time step, the shaded box contains the pair $\Phi_G \mid W_G + 1$ for the group G before it is selected.

the current group to the largest weight group G_1 and run client C_1 . Based on (1), C_1 would be run for three time quanta before selecting G_2 again to run client C_2 . After running C_2 the second time, W_2 would increase such that $\frac{W_2+1}{W_3+1} = 3 > \frac{\Phi_2}{\Phi_3} = 2$, so GR^3 would then move on to the last group G_3 and run client C_3 . The resulting schedule would then be: $G_1, G_1, G_2, G_1, G_1, G_1, G_2, G_3$. Each group therefore receives its proportional allocation in accordance with its respective group weight.

The GR^3 **intragroup scheduling** algorithm selects a client from the selected group. All clients within a group have weights within a factor of two, and all client weights in a group G are normalized with respect to the minimum possible weight, $\phi_{\min}^G = 2^{\sigma_G}$, for any client in the group. GR^3 then effectively traverses through a group's queue in round-robin order, allocating each client its normalized weight worth of time quanta. GR^3 keeps track of subunitary fractional time quanta that cannot be used and accumulates them in a deficit value for each client. Hence, each client is assigned either one or two time quanta, based on the client's normalized weight and its previous allocation.

More specifically, the GR^3 intragroup scheduler considers the scheduling of clients in rounds. A *round* is one pass through a group G 's run queue of clients from beginning to end. The group run queue does not need to be sorted in any manner. During each round, the GR^3 intragroup algorithm considers the clients in round-robin order and executes the following simple routine:

INTRAGROUP-SCHEDULE(G)

```

1  $C \leftarrow G[k]$   $\triangleright k$  is the current position in the round
2 if  $D_C < 1$ 
3   then  $k \leftarrow (k + 1) \bmod |G|$ 
4    $C \leftarrow G[k]$ 
5    $D_C \leftarrow D_C + \phi_C / \phi_{\min}^G$ 
6  $D_C \leftarrow D_C - 1$ 
7 return  $C$ 
```

For each runnable client C , the scheduler determines the maximum number of time quanta that the client can be selected to run in this round as $\lfloor \frac{\phi_C}{\phi_{\min}^G} + D_C(r-1) \rfloor$. $D_C(r)$, the deficit of client C after round r , is the time quantum fraction left over after round r : $D_C(r) = \frac{\phi_C}{\phi_{\min}^G} + D_C(r -$

$1) - \lfloor \frac{\phi_C}{\phi_{\min}^G} + D_C(r-1) \rfloor$, with $D_C(0) = \frac{\phi_C}{\phi_{\min}^G}$. Thus, in each round, C is allotted one time quantum plus any additional leftover from the previous round, and $D_C(r)$ keeps track of the amount of service that C missed because of rounding down its allocation to whole time quanta. We observe that $0 \leq D_C(r) < 1$ after any round r so that any client C will be allotted one or two time quanta. Note that if a client is allotted two time quanta, it first executes for one time quantum and then executes for the second time quantum the next time the intergroup scheduler selects its respective group again (in general, following a timespan when clients belonging to other groups get to run).

To illustrate how GR^3 works with intragroup scheduling, Figure 2 shows an example with six clients C_1 through C_6 with weights 12, 3, 3, 2, 2, and 2, respectively. The six clients will be put in two groups G_1 and G_2 with respective group order 1 and 3 as follows: $G_1 = \{C_2, C_3, C_4, C_5, C_6\}$ and $G_2 = \{C_1\}$. The weight of the groups are $\Phi_1 = \Phi_2 = 12$. GR^3 intergroup scheduling will consider the groups in this order: $G_1, G_2, G_1, G_2, G_1, G_2, G_1, G_2, G_1, G_2, G_1, G_2$. G_2 will schedule client C_1 every time G_2 is considered for service since it has only one client. Since $\phi_{\min}^{G_1} = 2$, the normalized weights of clients C_2, C_3, C_4, C_5 , and C_6 are 1.5, 1.5, 1, 1, and 1, respectively. In the beginning of round 1 in G_1 , each client starts with 0 deficit. As a result, the intragroup scheduler will run each client in G_1 for one time quantum during round 1. After the first round, the deficit for C_2, C_3, C_4, C_5 , and C_6 are 0.5, 0.5, 0, 0, and 0. In the beginning of round 2, each client gets another $\phi_i / \phi_{\min}^{G_1}$ allocation, plus any deficit from the first round. As a result, the intragroup scheduler will select clients C_2, C_3, C_4, C_5 , and C_6 to run in order for 2, 2, 1, 1, and 1 time quanta, respectively, during round 2. The resulting schedule would then be: $C_2, C_1, C_3, C_1, C_4, C_1, C_5, C_1, C_6, C_1, C_2, C_1, C_2, C_1, C_3, C_1, C_3, C_1, C_4, C_1, C_5, C_1, C_6, C_1$.

2.3 GR^3 Dynamic Considerations

We now discuss how GR^3 allows clients to be dynamically created, terminated, or change run state. Runnable clients can be selected for execution by the scheduler, while clients that are not runnable cannot. With no loss of generality, we assume that a client is created before it can become runnable, and a client becomes not runnable before it is terminated. As a result, client creation and termination have no effect on the GR^3 run queues.

When a client C with weight ϕ_C becomes runnable, it is inserted into group $G = G(C)$ such that ϕ_C is between 2^{σ_G} and $2^{\sigma_G+1} - 1$. If the group was previously empty, a new group is created, the client becomes the current client of the group, and g , the number of groups, is incremented. If the group was not previously empty, GR^3 inserts the client into the respective group's run queue right before the current client; it will be serviced after all of the other clients

P	Number of processors.
ϕ^k	Processor k .
$C(\phi)$	Client running on processor ϕ .
F_C	Frontlog for client C .

Table 2: GR^3MP terminology

processor needs to be scheduled, GR^3MP selects the client that would run next under GR^3 , essentially scheduling multiple processors from its central run queue as GR^3 schedules a single processor. However, there is one obstacle to simply applying a uniprocessor algorithm on a multiprocessor system. Each client can only run on one processor at any given time. As a result, GR^3MP cannot select a client to run that is already running on another processor even if GR^3 would schedule that client in the uniprocessor case. For example, if GR^3 would schedule the same client consecutively, GR^3MP cannot schedule that client consecutively on another processor if it is still running.

To handle this situation while maintaining fairness, GR^3MP introduces the notion of a **frontlog**. The frontlog F_C for some client C running on a processor ϕ^k ($C = C(\phi^k)$) is defined as the number of time quanta for C accumulated as C gets selected by GR^3 and cannot run because it is already running on ϕ^k . The frontlog F_C is then queued up on ϕ^k .

Given a client that would be scheduled by GR^3 but is already running on another processor, GR^3MP uses the frontlog to assign the client a time quantum now but defer the client's use of it until later. Whenever a processor finishes running a client for a time quantum, GR^3MP checks whether the client has a non-zero frontlog, and, if so, continues running the client for another time quantum and decrements its frontlog by one, without consulting the central queue. The frontlog mechanism not only ensures that a client receives its proportional share allocation, it also takes advantage of any cache affinity by continuing to run the client on the same processor.

When a processor finishes running a client for a time quantum and its frontlog is zero, we call the processor *idle*. GR^3MP schedules a client to run on the idle processor by performing a GR^3 scheduling decision on the central queue. If the selected client is already running on some other processor, we increase its frontlog and repeat the GR^3 scheduling, each time incrementing the frontlog of the selected client, until we find a client that is not currently running. We assign this client to the idle processor for one time quantum. This description assumes that there are least $P+1$ clients in the system. Otherwise, scheduling is easy: an idle processor will either run the client it just ran, or idles until more clients arrive. In effect, each client will simply be assigned its own processor. Whenever a processor needs to perform a scheduling decision, it thus executes the following routine:

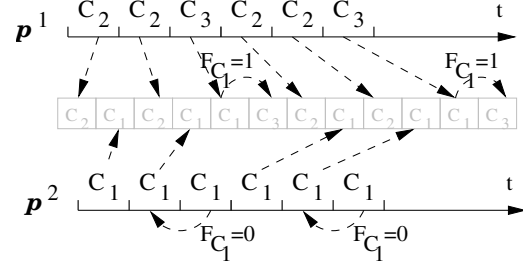


Figure 3: GR^3 multiprocessor scheduling. The two processors schedule either from the central queue, or use the frontlog mechanism when the task is already running.

MP-SCHEDULE(ϕ^k)

```

1   $C \leftarrow C(\phi^k)$                                  $\triangleright$  Client just run
2  if  $C = \text{NIL}$ 
3      then if  $N < P$ 
4          then return NIL                             $\triangleright$  Idle
5      else if  $F_C > 0$ 
6          then  $F_C \leftarrow F_C - 1$ 
7          return  $C$ 
8   $C \leftarrow \text{INTERGROUP-SCHEDULE}()$ 
9  while  $\exists \phi$  s.t.  $C = C(\phi)$ 
10     do  $F_C \leftarrow F_C + 1$ 
11      $C \leftarrow \text{INTERGROUP-SCHEDULE}()$ 
12 return  $C$ 

```

To illustrate GR^3MP scheduling, Figure 3 shows an example on a dual-processor system with three clients C_1 , C_2 , and C_3 of weights 3, 2, and 1, respectively. C_1 and C_2 will then be part of the order 1 group (assume C_2 is before C_1 in the round-robin queue of this group), whereas C_3 is part of the order 0 group. The GR^3 schedule is C_2 , C_1 , C_2 , C_1 , C_1 , C_3 . ϕ^1 will then select C_2 to run, and ϕ^2 selects C_1 . When ϕ^1 finishes, according to GR^3 , it will select C_2 once more, whereas ϕ^2 selects C_1 again. When ϕ^1 again selects the next GR^3 client, which is C_1 , it finds that it is already running on ϕ^2 and thus we set $F_{C_1} = 1$ and select the next client, which is C_3 , to run on ϕ^1 . When ϕ^2 finishes running C_1 for its second time quantum, it finds $F_{C_1} = 1$, sets $F_{C_1} = 0$ and continues running C_1 without any scheduling decision on the GR^3 queue.

3.2 GR^3MP Dynamic Considerations

GR^3MP basically does the same thing as the GR^3 algorithm under dynamic considerations. However, the frontlogs used in GR^3MP need to be accounted for appropriately. If some processors have long frontlogs for their currently running clients, newly arriving clients may not be run by those processors until their frontlogs are processed, resulting in bad responsiveness for the new clients. Although in between any two client arrivals or departures, some processors must have no frontlog, the set of such

processors can be as small as a single processor. In this case, newly arrived clients will end up competing with other clients already in the run queue only for those few processors, until the frontlog on the other processors is exhausted.

GR^3MP provides fair and responsive allocations by creating frontlogs for newly arriving clients. Each new client is assigned a frontlog equal to a fraction of the total current frontlog in the system based on its proportional share. Each processor now maintains a queue of frontlog clients and a new client with a frontlog is immediately assigned to one of the processor frontlog queues. Rather than running its currently running client until it completes its frontlog, each processor now round robins among clients in its frontlog queue. Given that frontlogs are small in practice, round-robin scheduling is used for frontlog clients for its simplicity and fairness. GR^3MP balances the frontlog load on the processors by placing new frontlog clients on the processor with the smallest frontlog summed across all its frontlog clients.

More precisely, whenever a client C arrives, and it belongs in group $G(C)$, GR^3MP performs the same group operations as in the single processor GR^3 algorithm. GR^3MP finds the processor \wp^k with the smallest frontlog, then creates a frontlog for client C on \wp^k of length $F_C = F_T \frac{\phi_C}{\Phi_T}$, where F_T is the total frontlog on all the processors. Let $C' = C(\wp^k)$. Then, assuming no further clients arrive, \wp^k will round-robin between C and C' and run C for F_C and C' for $F_{C'}$ time quanta.

When a client becomes not runnable, GR^3MP uses the same lazy removal mechanism used in GR^3 . If it is removed from the run queue and has a frontlog, GR^3MP simply discards it since each client is assigned a frontlog based on the current state of the system when it becomes runnable again.

3.3 GR^3MP Weight Readjustment

Since no client can run on more than one processor at a time, no client can consume more than a $1/P$ fraction of the processing in a multiprocessor system. A client C with weight ϕ_C greater than Φ_T/P is considered *infeasible* since it cannot receive its proportional share allocation ϕ_C/Φ_T without using more than one processor simultaneously. GR^3MP should then give the client its maximum possible service, and simply assign such a client its own processor to run on. However, since the scheduler uses client weights to determine which client to run, an infeasible client's weight must be adjusted so that it is feasible to ensure that the scheduling algorithm runs correctly to preserve fairness (assuming there are at least P clients). GR^3MP potentially needs to perform weight readjustment whenever a client is inserted or removed from the run queue to make sure that all weights are feasible.

To understand the problem of weight readjustment, consider the sequence of all clients, ordered by weight:

$S_{1,N} = C_1, C_2, \dots, C_N$ with $\phi_1 \geq \phi_2 \geq \dots \geq \phi_N$. We call the subsequence $S_{k,N} = C_k, C_{k+1}, \dots, C_N$ *Q-feasible*, if $\phi_k \leq \frac{1}{Q} \sum_{j=k}^N \phi_j$.

Lemma 1. *The client mix in the system is feasible if and only if $S_{1,N}$ is P -feasible.*

Proof. If $\phi_1 > \frac{\Phi_T}{P}$, C_1 is infeasible, so the mix is infeasible. Conversely, if $\phi_1 \leq \frac{\Phi_T}{P}$, then for any client C_l , $\phi_l \leq \phi_1 \leq \frac{\Phi_T}{P}$, implying all clients are feasible. The mix is then feasible $\iff \phi_1 \leq \frac{\Phi_T}{P} = \frac{1}{P} \sum_{j=1}^N \phi_j$, or, equivalently, $S_{1,N}$ is P -feasible. \square

Lemma 2. *$S_{k,N}$ is Q -feasible $\Rightarrow S_{k+1,N}$ is $(Q-1)$ -feasible.*

Proof. $\phi_k \leq \frac{1}{Q} \sum_{j=k}^N \phi_j \iff Q\phi_k \leq \phi_k + \sum_{j=k+1}^N \phi_j \iff \phi_k \leq \frac{1}{Q-1} \sum_{j=k+1}^N \phi_j$. Since $\phi_{k+1} \leq \phi_k$, the lemma follows. \square

The feasibility problem is then to identify the least k (denoted the *feasibility threshold*, f) such that $S_{k,N}$ is $(P-k+1)$ -feasible. If $f = 1$, then the client mix is feasible. Otherwise, the *infeasible set* $S_{1,f-1} = C_1, \dots, C_{f-1}$ contains the infeasible clients, whose weight needs to be scaled down to $1/P$ of the resulting total weight. The cardinality $f-1$ of the infeasible set is less than P . However, the sorted sequence $S_{1,N}$ is expensive to maintain, such that traversing it and identifying the feasibility threshold is not an efficient solution.

GR^3MP leverages its grouping strategy to perform fast weight readjustment. GR^3MP starts with the unmodified client weights, finds the set I of infeasible clients, and adjust their weights to be feasible. To construct I , the algorithm traverses the list of groups in decreasing order of their group order σ_G , until it finds a group not all of whose clients are infeasible. We denote by $|I|$ the cardinality of I and by Φ_I the sum of weights of the clients in I , $\sum_{C \in I} \phi_C$. The GR^3MP weight readjustment algorithm is as follows:

WEIGHT-READJUSTMENT()

```

1  RESTORE-ORIGINAL-WEIGHTS
2   $I \leftarrow \emptyset$ 
3   $G \leftarrow$  greatest order group
4  while  $|G| < P - |I|$  and  $2^{\sigma_G} > \frac{\Phi_T - \Phi_I - \Phi_G}{P - |I| - |G|}$ 
5      do  $I \leftarrow I \cup G$ 
6           $G \leftarrow \text{next}(G)$   $\triangleright$  by group order
7  if  $|G| < 2(P - |I|)$ 
8      then  $I \leftarrow I \cup \text{INFEASIBLE}(G, P - |I|, \Phi_T - \Phi_I)$ 
9   $\Phi_T^f \leftarrow \Phi_T - \Phi_I$ 
10  $\Phi_T \leftarrow \frac{P}{P - |I|} \Phi_T^f$ 
11 for each  $C \in I$ 
12     do  $\phi_C \leftarrow \frac{\Phi_T}{P}$ 
```

The correctness of the algorithm is based on Lemma 2. Let some group G span the subsequence $S_{i,j}$ of the sequence of ordered clients $S_{1,N}$. Then $2^{\sigma_{G+1}} - 1 \geq \phi_i \geq \dots \geq \phi_j \geq 2^{\sigma_G}$ and it is easy to show:

- $2^{\sigma_G} > \frac{\Phi_T - \Phi_I - \Phi_G}{P - |I| - |G|} \Rightarrow j < f$ (all clients in $S_{1,j}$ are infeasible).
- $2^{\sigma_G} \leq \frac{\Phi_T - \Phi_I - \Phi_G}{P - |I| - |G|} \Rightarrow j + 1 \geq f$ (all clients in $S_{j+1,N}$ are feasible).

Once we reach line 7, we know $S_{j+1,N}$ is $(P - j)$ -feasible, and $i \leq f \leq j + 1$. If $|G| \geq 2(P - |I|)$, GR^3MP can stop searching for infeasible clients since all clients $C \in G$ are feasible, and $f = i$ (equivalently, $S_{i,N}$ is $(P - |I|)$ -feasible): $\phi_C < 2^{\sigma_{G+1}} \leq 2 \frac{1}{|G|} \Phi_G \leq \frac{1}{P - |I|} \Phi_G \leq \frac{1}{P - |I|} (\Phi_T - \Phi_I)$. Otherwise, if $|G| < 2(P - |I|)$, then $i < f \leq j + 1$ and GR^3MP needs to search through G to determine which clients are infeasible (equivalently, find f). Since the number of clients in G is small, we can sort all clients in G by weight. Then, starting from the largest weight client in G , find the first feasible client. A simple algorithm is then the following:

INFEASIBLE(G, Q, Φ)

```

1   $I \leftarrow \emptyset$ 
2  for each  $C \in G$  in sorted order
3      do if  $\phi_C > \frac{1}{Q - |I|} (\Phi - \Phi_I)$ 
4          then  $I \leftarrow I \cup \{C\}$ 
5          else return  $I$ 
6  return  $I$ 
```

GR^3MP can alternatively use a more complicated but lower time complexity divide-and-conquer algorithm to find the infeasible clients in G . In this case, GR^3MP partitions G around its median \bar{C} into G_S , the set of G clients that have weight less than $\phi_{\bar{C}}$ and G_B , the set of G clients that have weight larger than $\phi_{\bar{C}}$. By Lemma 2, if \bar{C} is feasible, $G_S \cup \{\bar{C}\}$ is feasible, and we recurse on G_B . Otherwise, all clients in $G_B \cup \{\bar{C}\}$ are infeasible, and we recurse on G_S to find all infeasible clients. The algorithm finishes when the set we need to recurse on is empty:

INFEASIBLE(G, Q, Φ)

```

1  if  $G = \emptyset$ 
2      then return  $\emptyset$ 
3   $\bar{C} \leftarrow \text{MEDIAN}(G)$ 
4   $(G_S, G_B) \leftarrow \text{PARTITION}(G, \phi_{\bar{C}})$ 
5  if  $\phi_{\bar{C}} > \frac{\Phi - \Phi_{G_B}}{Q - |G_B|}$ 
6      then return  $G_B \cup \{\bar{C}\} \cup$ 
            $\text{INFEASIBLE}(G_S, Q - |G_B| - 1, \Phi - \Phi_{G_B} - \phi_{\bar{C}})$ 
7  else return  $\text{INFEASIBLE}(G_B, Q, \Phi)$ 
```

Once all infeasible clients have been identified, $\text{WEIGHT-READJUSTMENT}()$ determines the sum of the

weights of all feasible clients, $\Phi_T^f = \Phi_T - \Phi_I$. We can now compute the new total weight in the system as $\Phi_T = \frac{P}{P - |I|} \Phi_T^f$, namely the solution to the equation $\Phi_T^f + |I| \frac{x}{P} = x$. Once we have the adjusted Φ_T , we change all the weights for the infeasible clients in I to $\frac{\Phi_T}{P}$. Lemma 6 in Section 4.2 shows the readjustment algorithm runs in time $O(P)$ and is thus asymptotically optimal, since there can be $\Theta(P)$ infeasible clients.

4 GR^3 Fairness and Complexity

We analyze the fairness and complexity of GR^3 and GR^3MP . To analyze fairness, we use a more formal notion of proportional fairness defined as *service error*, a measure widely used [1, 7, 9, 17, 18, 19, 25, 27] in the analysis of scheduling algorithms. To simplify the analysis, we will assume that clients are always runnable and derive fairness bounds for such a case. Subsequently, we address the impact of arrivals and departures.

We use a strict measure of service error (equivalent in this context to the *Normalized Worst-case Fair Index* [1]) relative to Generalized Processor Sharing (GPS) [16], an idealized model that achieves *perfect fairness*: $w_C = W_T \frac{\phi_C}{\Phi_T}$, an ideal state in which each client C always receives service exactly proportional to its weight. Although all real-world schedulers must time-multiplex resources in time units of finite size and thus cannot maintain perfect fairness, some algorithms stay closer to perfect fairness than others and therefore have less service error. We quantify how close an algorithm gets to perfect fairness using the *client service time error*, which is the difference between the service received by client C and its share of the total work done by the processor: $e_C = w_C - W_T \frac{\phi_C}{\Phi_T}$. A positive service time error indicates that a client has received more than its ideal share over a time interval; a negative error indicates that it has received less. To be precise, the error e_C measures how much time a client C has received beyond its ideal allocation. A proportional share scheduler should minimize the absolute value of the allocation error of all clients with minimal scheduling overhead.

We provide bounds on the service error of GR^3 and GR^3MP . To do this, we define two other measures of service error. The *group service time error* is a similar measure for groups that quantifies the fairness of allocating the processor among groups: $E_G = W_G - W_T \frac{\Phi_G}{\Phi_T}$. The *group-relative service time error* represents the service time error of client C if there were only a single group $G = G(C)$ in the scheduler and is a measure of the service error of a client with respect to the work done on behalf of its group: $e_{C,G} = w_C - W_G \frac{\phi_C}{\Phi_G}$. We first show bounds on the group service error of the intergroup scheduling algorithm. We then show bounds on the group-relative service error of the intragroup scheduling algorithm. We combine these results to obtain the overall client service error bounds. We also discuss the scheduling overhead of GR^3 and GR^3MP in

terms of their time complexity. We show that both algorithms can make scheduling decisions in $O(1)$ time with $O(1)$ service error given a constant number of groups. Due to space constraints, most of the proofs are omitted. Further proof details are available in [5].

4.1 Analysis of GR^3

Intergroup Fairness For the case when the weight ratios of consecutive groups in the group list are integers, we get the following:

Lemma 3. *If $\frac{\Phi_j}{\Phi_{j+1}} \in \mathbb{N}$, $1 \leq j < g$, then $-1 < E_{G_k} \leq (g - k) \frac{\Phi_k}{\Phi_T}$ for any group G_k .*

Proof sketch: If the group currently scheduled is G_k , then the work to weight ratio of all groups G_j , $j < k$, is the same. For $j > k$, $\frac{W_{j+1}}{\Phi_{j+1}} \leq \frac{W_j}{\Phi_j} \leq \frac{W_{j+1}+1}{\Phi_{j+1}} - \frac{1}{\Phi_j}$ as a consequence of the well-ordering condition (2). After some rearrangements, we can sum over all j and bound W_k , and thus E_{G_k} above and below. The additive $\frac{1}{\Phi_j}$ will cause the $g - 1$ upper bound.

In the general case, we get similar, but slightly weaker bounds.

Lemma 4. *For any group G_k , $-\frac{(g-k)(g-k-1)}{2} \frac{\Phi_k}{\Phi_T} - 1 < E_{G_k} < g - 1$.*

The proof for this case (omitted) follows reasoning similar to that of the previous lemma, but with several additional complications.

It is clear that the lower bound is minimized when setting $k = 1$. Thus, we have

Corollary 1. $-\frac{(g-1)(g-2)}{2} \frac{\Phi_G}{\Phi_T} - 1 < E_G < g - 1$ for any group G .

Intragroup Fairness Within a group, all weights are within a factor of two and the group-relative error is bound by a small constant. The only slightly subtle point is to deal with fractional rounds.

Lemma 5. $-3 < e_{C,G} < 4$ for any client $C \in G$.

Overall Fairness of GR^3 Based on the identity $e_C = e_{C,G} + \frac{\phi_C}{\Phi_G} E_G$ which holds for any group G and any client $C \in G$, we can combine the inter- and intragroup analyses to bound the overall fairness of GR^3 .

Theorem 1. $-\frac{(g-1)(g-2)}{2} \frac{\phi_C}{\Phi_T} - 4 < e_C < g + 3$ for any client C .

The negative error of GR^3 is thus bounded by $O(g^2)$ and the positive error by $O(g)$. Recall, g , the number of groups, does not depend on the number of clients in the system.

Dynamic Fairness of GR^3 We can consider a client arrival or removal as an operation where a group is first removed from the group list and added in a different place with a different weight. We argue that fairness is preserved by these operations: when group G_k is removed, then G_{k-1} , G_k , and G_{k+1} were well-ordered as defined in (2), so after the removal, G_{k-1} and G_{k+1} , now neighbors, will be well-ordered by transitivity. When a group, call it $G_{i+(1/2)}$, is inserted between G_i and G_{i+1} , it can be proven that the work readjustment formula in Section 2.3 ensures $G_{i+(1/2)}$ and G_{i+1} are well-ordered. In the case of G_i and $G_{i+(1/2)}$, we can show that we can achieve well-ordering by running $G_{i+(1/2)}$ at most one extra time. Thus, modulo this readjustment, the intragroup algorithm's fairness bounds are preserved. An important property of our algorithm that follows is that the pairwise ratios of work of clients *not* part of the readjusted group will be unaffected. Since the intragroup algorithm has constant fairness bounds, the disruption for the work received by clients inside the adjusted group is only $O(1)$.

Time Complexity GR^3 manages to bound its service error by $O(g^2)$ while maintaining a strict $O(1)$ scheduling overhead. The intergroup scheduler either selects the next group in the list, or reverts to the first one, which takes constant time. The intragroup scheduler is even simpler, as it just picks the next client to run from the unordered round robin list of the group. Adding and removing a client is worst-case $O(g)$ when a group needs to be relocated in the ordered list of groups. This could of course be done in $O(\log g)$ time (using binary search, for example), but the small value of g in practice does not justify a more complicated algorithm.

The *space complexity* of GR^3 is $O(g) + O(N) = O(N)$. The only additional data structure beyond the unordered lists of clients is an ordered list of length g to organize the groups.

4.2 Analysis of GR^3MP

Overall Fairness of GR^3MP Given feasible client weights after weight readjustment, the service error for GR^3MP is bounded below by the GR^3 error, and above by a bound which improves with more processors.

Theorem 2. $-\frac{(g-1)(g-2)}{2} \frac{\phi_C}{\Phi_T} - 4 < e_C < 2g + 10 + \frac{(g-1)(g-2)}{2P}$ for any client C .

Time Complexity of GR^3MP The frontlogs create an additional complication when analyzing the time complexity of GR^3MP . When an idle processor looks for its next client, it runs the simple $O(1)$ GR^3 algorithm to find a client C . If C is not running on any other processor, we are done, but otherwise we place it on the frontlog and then we must rerun the GR^3 algorithm until we find a client

that is not running on any other processor. Since for each such client, we increase its allocation on the processor it runs, the amortized time complexity remains $O(1)$. The upper bound on the time that any single scheduling decision takes is given by the maximum length of any scheduling sequence of GR^3 consisting of only some fixed subset of $P - 1$ clients.

Theorem 3. *The time complexity per scheduling decision in GR^3MP is bounded above by $\frac{(g-k)(g-k+1)}{2} + (k + 1)(g - k + 1)P + 1$ where $1 \leq k \leq g$.*

Thus, the length of any schedule consisting of at most $P - 1$ clients is $O(g^2P)$. Even when a processor has frontlogs for several clients queued up on it, it will schedule in $O(1)$ time, since it performs round-robin among the frontlogged clients. Client arrivals and departures take $O(g)$ time because of the need to readjust group weights in the saved list of groups. Moreover, if we also need to use the weight readjustment algorithm, we incur an additional $O(P)$ overhead on client arrivals and departures.

Lemma 6. *The complexity of the weight readjustment algorithm is $O(P)$.*

Proof. Restoring the original weights will worst case touch a number of groups equal to the number of previously infeasible clients, which is $O(P)$. Identifying the infeasible clients involves iterating over at most P groups in decreasing sequence based on group order, as described in Section 3.3. For the last group considered, we only attempt to partition it into feasible and infeasible clients of its size is less than $2P$. Since partitioning of a set can be done in linear time, and we recurse on a subset half the size, this operation is $O(P)$ as well. \square

For small P , the $O(P \log(P))$ sorting approach to determine infeasible clients in the last group considered is simpler and in practice performs better than the $O(P)$ recursive partitioning. Finally, altering the active group structure to reflect the new weights is a $O(P + g)$ operation, as two groups may need to be re-inserted in the ordered list of groups.

5 Measurements and Results

We have implemented GR^3 uniprocessor and multiprocessor schedulers in the Linux operating system and measured their performance. We present some experimental data quantitatively comparing GR^3 performance against other popular scheduling approaches from both industrial practice and research. We have conducted both extensive simulation studies and detailed measurements of real kernel scheduler performance on real applications.

Section 5.1 presents simulation results comparing the proportional sharing accuracy of GR^3 and GR^3MP against WRR, WFQ [18], SFQ [13], VTRR [17], and

SRR [9]. The simulator enabled us to isolate the impact of the scheduling algorithms themselves and examine the scheduling behavior of these different algorithms across hundreds of thousands of different combinations of clients with different weight values.

Section 5.2 presents detailed measurements of real kernel scheduler performance by comparing our prototype GR^3 Linux implementation against the standard Linux scheduler, a WFQ scheduler, and a VTRR scheduler. The experiments we have done quantify the scheduling overhead and proportional share allocation accuracy of these schedulers in a real operating system environment under a number of different workloads.

All our kernel scheduler measurements were performed on an IBM Netfinity 4500 system with one or two 933 MHz Intel Pentium III CPUs, 512 MB RAM, and 9 GB hard drive. The system was installed with the Debian GNU/Linux distribution version 3.0 and all schedulers were implemented using Linux kernel version 2.4.19. The measurements were done by using a minimally intrusive tracing facility that writes timestamped event identifiers into a memory log and takes advantage of the high-resolution clock cycle counter available with the Intel CPU, providing measurement resolution at the granularity of a few nanoseconds. Getting a timestamp simply involved reading the hardware cycle counter register. We measured the timestamp overhead to be roughly 35 ns per event.

The kernel scheduler measurements were performed on a fully functional system. All experiments were performed with all system functions running and the system connected to the network. At the same time, an effort was made to eliminate variations in the test environment to make the experiments repeatable.

5.1 Simulation Studies

We built a scheduling simulator that measures the service time error, described in Section 4, of a scheduler on a set of clients. The simulator takes four inputs, the scheduling algorithm, the number of clients N , the total sum of weights Φ_T , and the number of client-weight combinations. The simulator randomly assigns weights to clients and scales the weights to ensure that they add up to Φ_T . It then schedules the clients using the specified algorithm as a real scheduler would, assuming no client blocks, and tracks the resulting service time error. The simulator runs the scheduler until the resulting schedule repeats, then computes the maximum (most positive) and minimum (most negative) service time error across the nonrepeating portion of the schedule for the given set of clients and weight assignments. This process is repeated for the specified number of client-weight combinations. We then compute the maximum service time error and minimum service time error for the specified number of client-weight combinations to obtain a “worst-case” error range.

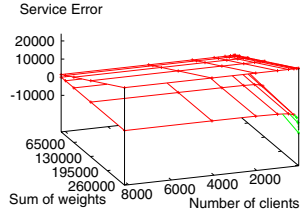


Figure 4: WRR error

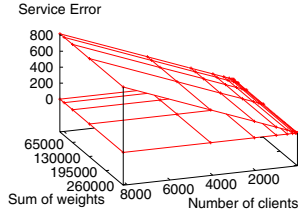


Figure 5: WFQ error

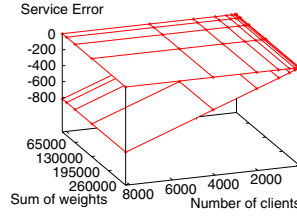


Figure 6: SFQ error

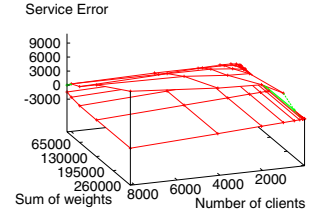


Figure 7: VTRR error

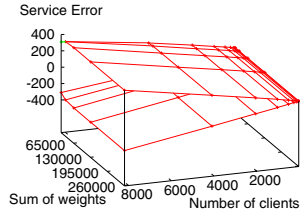


Figure 8: SRR error

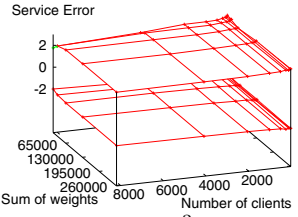


Figure 9: GR^3 error

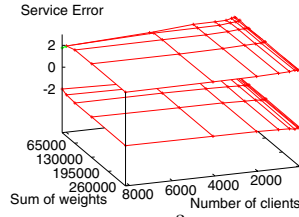


Figure 10: GR^3MP error

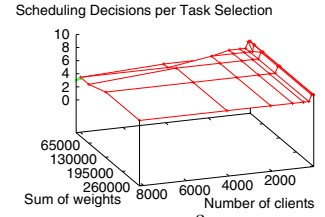


Figure 11: GR^3MP overhead

To measure proportional fairness accuracy, we ran simulations for each scheduling algorithm on 45 different combinations of N and Φ_T (32 up to 8192 clients and 16384 up to 262144 total weight, respectively). Since the proportional sharing accuracy of a scheduler is often most clearly illustrated with skewed weight distributions, one of the clients was given a weight equal to 10 percent of Φ_T . All of the other clients were then randomly assigned weights to sum to the remaining 90 percent of Φ_T . For each pair (N, Φ_T) , we ran 2500 client-weight combinations and determined the resulting worst-case error range.

The worst-case service time error ranges for WRR, WFQ, SFQ, VTRR, SRR, and GR^3 with these skewed weight distributions are in Figures 4 to 9. Due to space constraints, WF²Q error is not shown since the results simply verify its known mathematical error bounds of -1 and 1 tu. Each figure consists of a graph of the error range for the respective scheduling algorithm. Each graph shows two surfaces representing the maximum and minimum service time error as a function of N and Φ_T for the same range of values of N and Φ_T . Figure 4 shows WRR's service time error is between -12067 tu and 23593 tu. Figure 5 shows WFQ's service time error is between -1 tu and 819 tu, which is much less than WRR. Figure 6 shows SFQ's service time error is between -819 tu and 1 tu, which is almost a mirror image of WFQ. Figure 7 shows VTRR's service error is between -2129 tu and 10079 tu. Figure 8 shows SRR's service error is between -369 tu and 369 tu.

In comparison, Figure 9 shows the service time error for GR^3 only ranges from -2.5 to 3.0 tu. GR^3 has a smaller error range than all of the other schedulers measured except WF²Q. GR^3 has both a smaller negative and smaller positive service time error than WRR, VTRR, and SRR. While GR^3 has a much smaller positive service error than WFQ, WFQ does have a smaller negative service

time error since it is bounded below at -1 . Similarly, GR^3 has a much smaller negative service error than SFQ, though SFQ's positive error is less since it is bounded above at 1 . Considering the total service error range of each scheduler, GR^3 provides well over two orders of magnitude better proportional sharing accuracy than WRR, WFQ, SFQ, VTRR, and SRR. Unlike the other schedulers, these results show that GR^3 combines the benefits of low service time errors with its ability to schedule in $O(1)$ time.

Note that as the weight skew becomes more accentuated, the service error can grow dramatically. Thus, increasing the skew from 10 to 50 percent results in more than a fivefold increase in the error magnitude for SRR, WFQ, and SFQ, and also significantly worse errors for WRR and VTRR. In contrast, the error of GR^3 is still bounded by small constants: -2.3 and 4.6 .

We also measured the service error of GR^3MP using this simulator configured for an 8 processor system, where the weight distribution was the same as for the uniprocessor simulations above. Note that the client given 0.1 of the total weight was feasible, since $0.1 < \frac{1}{8} = 0.125$. Figure 10 shows GR^3MP 's service error is between -2.5 tu and 2.8 tu, slightly better than for the uniprocessor case, a benefit of being able to run multiple clients in parallel. Figure 11 shows the maximum number of scheduling decisions that an idle processor needs to perform until it finds a client that is not running. This did not exceed seven, indicating that the number of decisions needed in practice is well below the worst-case bounds shown in Theorem 3.

5.2 Linux Kernel Measurements

To evaluate the scheduling overhead of GR^3 , we compare it against the standard Linux 2.4 scheduler, a WFQ scheduler, and a VTRR scheduler. Since WF²Q has the-

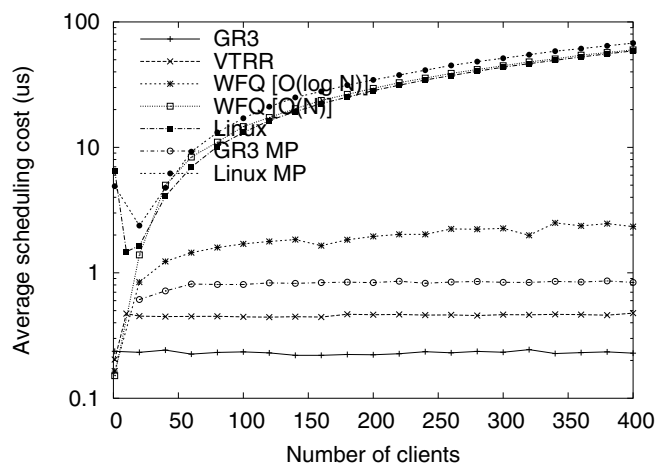


Figure 12: Average scheduling overhead

oretically the same time complexity as WFQ (but with larger constants, because of the complexity of its steps), we present WFQ as a lower bound for the overhead of WF²Q. We present results from several experiments that quantify how scheduling overhead varies as the number of clients increases. For the first experiment, we measure scheduling overhead for running a set of clients, each of which executed a simple micro-benchmark which performed a few operations in a while loop. A control program was used to fork a specified number of clients. Once all clients were runnable, we measured the execution time of each scheduling operation that occurred during a fixed time duration of 30 seconds. The measurements required two timestamps for each scheduling decision, so measurement error of 70 ns are possible due to measurement overhead. We performed these experiments on the standard Linux scheduler, WFQ, VTRR, and GR^3 for 1 to 400 clients.

Figure 12 shows the average execution time required by each scheduler to select a client to execute. Results for GR^3 , VTRR, WFQ, and Linux were obtained on uniprocessor system, and results for GR^3MP and LinuxMP were obtained running on a dual-processor system. Dual-processor results for WFQ and VTRR are not shown since MP-ready implementations of them were not available.

For this experiment, the particular implementation details of the WFQ scheduler affect the overhead, so we include results from two different implementations of WFQ. In the first, labeled “WFQ [$O(N)$]”, the run queue is implemented as a simple linked list which must be searched on every scheduling decision. The second, labeled “WFQ [$O(\log N)$]”, uses a heap-based priority queue with $O(\log N)$ insertion time. To maintain the heap-based priority queue, we used a fixed-length array. If the number of clients ever exceeds the length of the array, a costly array reallocation must be performed. Our initial array size was large enough to contain more than 400 clients, so this additional cost is not reflected in our measurements.

As shown in Figure 12, the increase in scheduling overhead as the number of clients increases varies a great deal between different schedulers. GR^3 has the smallest scheduling overhead. It requires roughly 300 ns to select a client to execute and the scheduling overhead is essentially constant for all numbers of clients. While VTRR scheduling overhead is also constant, GR^3 has less overhead because its computations are simpler to perform than the virtual time calculations required by VTRR. In contrast, the overhead for Linux and for $O(N)$ WFQ scheduling grows linearly with the number of clients. Both of these schedulers impose more than 200 times more overhead than GR^3 when scheduling a mix of 400 clients. $O(\log N)$ WFQ has much smaller overhead than Linux or $O(N)$ WFQ, but it still imposes significantly more overhead than GR^3 , with 8 times more overhead than GR^3 when scheduling a mix of 400 clients. Figure 12 also shows that GR^3MP provides the same $O(1)$ scheduling overhead on a multiprocessor, although the absolute time to schedule is somewhat higher due to additional costs associated with scheduling in multiprocessor systems. The results show that GR^3MP provides substantially lower overhead than the standard Linux scheduler, which suffers from complexity that grows linearly with the number of clients. Because of the importance of constant scheduling overhead in server systems, Linux has switched to Ingo Molnar’s $O(1)$ scheduler in the Linux 2.6 kernel. As a comparison, we also repeated this microbenchmark experiment with that scheduler and found that GR^3 still runs over 30 percent faster.

As another experiment, we measured the scheduling overhead of the various schedulers for *hackbench* [21], a Linux benchmark used for measuring scheduler performance with large numbers of processes entering and leaving the run queue at all times. It creates groups of readers and writers, each group having 20 reader tasks and 20 writer tasks, and each writer writes 100 small messages to each of the other 20 readers. This is a total of 2000 messages sent per writer, per group, or 40000 messages per group. We ran a modified version of *hackbench* to give each reader and each writer a random weight between 1 and 40. We performed these tests on the same set of schedulers for 1 group up to 100 groups. Using 100 groups results in up to 8000 processes running. Because *hackbench* frequently inserts and removes clients from the run queue, the cost of client insertion and removal is a more significant factor for this benchmark. The results show that the simple dynamic group adjustments described in Section 2.3 have low overhead, since $O(g)$ can be considered constant in practice.

Figure 13 shows the average scheduling overhead for each scheduler. The average overhead is the sum of the times spent on all scheduling events, selecting clients to run and inserting and removing clients from the run queue, divided by the number of times the scheduler selected a client to run. The overhead in Figure 13 is higher than the av-

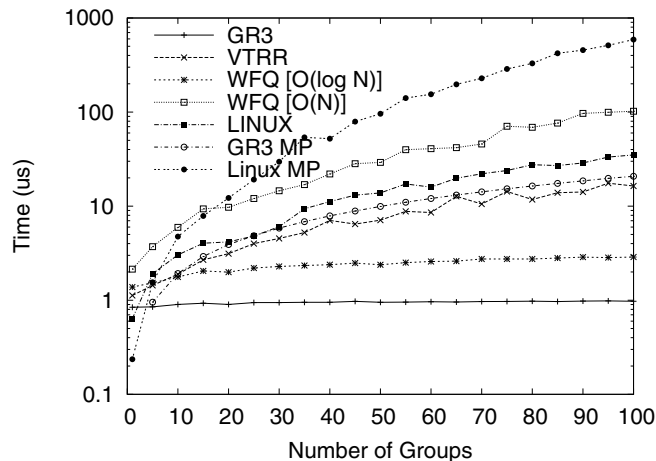


Figure 13: Hackbench weighted scheduling overhead

erage cost per schedule in Figure 12 for all the schedulers measured since Figure 13 includes a significant component of time due to client insertion and removal from the run queue. GR^3 still has by far the smallest scheduling overhead among all the schedulers measured. The overhead for GR^3 remains constant while the overhead for $O(\log N)$ WFQ, $O(N)$ WFQ, VTRR, and Linux grows with the number of clients. Client insertion, removal, and selection to run in GR^3 are independent of the number of clients. The cost for GR^3 is 3 times higher than before, with client selection to run, insertion, and removal each taking approximately 300 to 400 ns. For VTRR, although selecting a client to run is also independent of the number of clients, insertion overhead grows with the number of clients, resulting in much higher VTRR overhead for this benchmark.

To demonstrate GR^3 's efficient proportional sharing of resources on real applications, we briefly describe three simple experiments running web server workloads using the same set of schedulers: GR^3 and GR^3MP , Linux 2.4 uniprocessor and multiprocessor schedulers, WFQ, and VTRR. The web server workload emulates a number of virtual web servers running on a single system. Each virtual server runs the guitar music search engine used at guitarnotes.com, a popular musician resource web site with over 800,000 monthly users. The search engine is a perl script executed from an Apache mod-perl module that searches for guitar music by title or author and returns a list of results. The web server workload configured each server to pre-fork 100 processes, each running consecutive searches simultaneously.

We ran multiple virtual servers with each one having different weights for its processes. In the first experiment, we used six virtual servers, with one server having all its processes assigned weight 10 while all other servers had processes assigned weight 1. In the second experiment, we used five virtual servers and processes assigned to each server had respective weights of 1, 2, 3, 4, and 5. In the

third experiment, we ran five virtual servers which assigned a random weight between 1 and 10 to each process. For the Linux scheduler, weights were assigned by selecting `nice` values appropriately. Figures 14 to 19 present the results from the first experiment with one server with weight 10 processes and all other servers with weight 1 processes. The total load on the system for this experiment consisted of 600 processes running simultaneously. For illustration purposes, only one process from each server is shown in the figures. Conclusions drawn from the other experiments are the same; those results are omitted due to space constraints.

GR^3 and GR^3MP provided the best overall proportional fairness for these experiments while Linux provided the worst overall proportional fairness. Figures 14 to 17 show the amount of processor time allocated to each client over time for the Linux scheduler, WFQ, VTRR, and GR^3 . All of the schedulers except GR^3 and GR^3MP have a pronounced "staircase" effect for the search engine process with weight 10, indicating that CPU resources are provided in irregular bursts over a short time interval. For the applications which need to provide interactive responsiveness to web users, this can result in extra delays in system response time. It can be inferred from the smoother curves of Figure 17 that GR^3 and GR^3MP provide fair resource allocation at a finer granularity than the other schedulers.

6 Related Work

Round robin is one of the oldest, simplest and most widely used proportional share scheduling algorithms. Weighted round-robin (WRR) supports non-uniform client weights by running all clients with the same frequency but adjusting the size of their time quanta in proportion to their respective weights. Deficit round-robin (DRR) [22] was developed to support non-uniform service allocations in packet scheduling. These algorithms have low $O(1)$ complexity but poor short-term fairness, with service errors that can be on the order of the largest client weight in the system. GR^3 uses a novel variant of DRR for intragroup scheduling with $O(1)$ complexity, but also provides $O(1)$ service error by using its grouping mechanism to limit the effective range of client weights considered by the intragroup scheduler.

Fair-share schedulers [12, 14, 15] provide proportional sharing among users in a way compatible with a UNIX-style time-sharing framework based on multi-level feedback with a set of priority queues. These schedulers typically had low $O(1)$ complexity, but were often ad-hoc and could not provide any proportional fairness guarantees. Empirical measurements show that these approaches only provide reasonable proportional fairness over relatively large time intervals [12].

Lottery scheduling [26] gives each client a number of tickets proportional to its weight, then randomly selects a ticket. Lottery scheduling takes $O(\log N)$ time and relies on the law of large numbers for providing proportional fair-

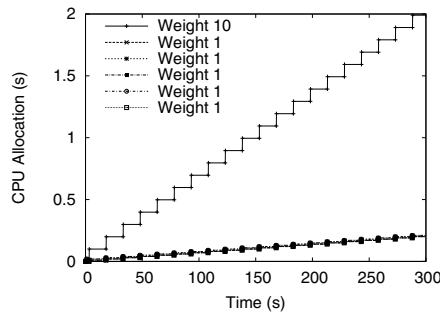


Figure 14: Linux uniprocessor

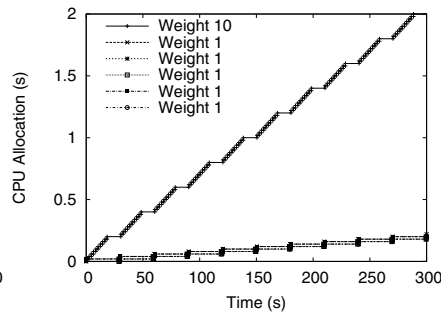


Figure 15: WFQ uniprocessor

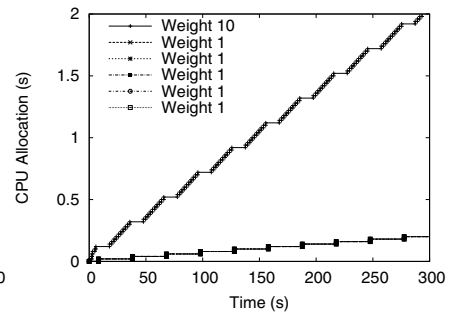


Figure 16: VTRR uniprocessor

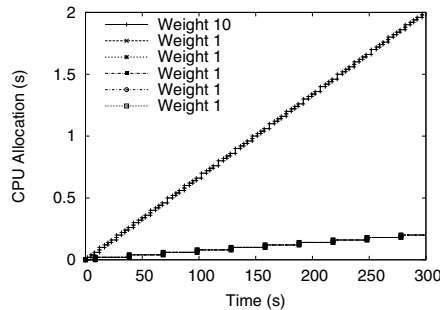


Figure 17: GR^3 uniprocessor

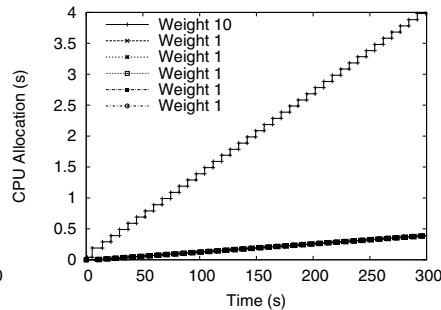


Figure 18: Linux multiprocessor

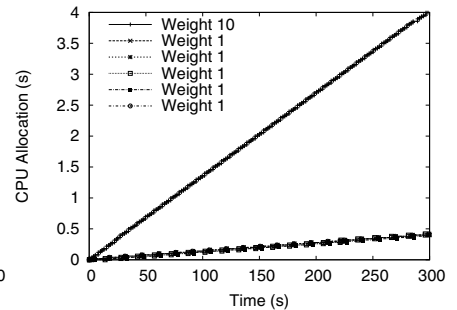


Figure 19: $GR^3 MP$ multiprocessor

ness. Thus, its allocation errors can be very large, typically much worse than WRR for clients with smaller weights.

Weighted Fair Queueing (WFQ) [11, 18], was first developed for network packet scheduling, and later applied to uniprocessor scheduling [26]. It assigns each client a virtual time and schedules the client with the earliest virtual time. Other fair queueing variants such as Virtual-clock [28], SFQ [13], SPFQ [24], and Time-shift FQ [10] have also been proposed. These approaches all have $O(\log N)$ time complexity, where N is the number of clients, because the clients must be ordered by virtual time. It has been shown that WFQ guarantees that the service time error for any client never falls below -1 [18]. However, WFQ can allow a client to get far ahead of its ideal allocation and accumulate a large positive service time error of $O(N)$, especially with skewed weight distributions.

Several fair queueing approaches have been proposed for reducing this $O(N)$ service time error. A hierarchical scheduling approach [26] reduces service time error to $O(\log N)$. Worst-Case Weighted Fair Queueing (WF²Q) [1] introduced eligible virtual times and can guarantee both a lower and upper bound on error of -1 and $+1$, respectively for network packet scheduling. It has also been applied to uniprocessor scheduling as Eligible Virtual Deadline First (EEVDF) [25]. These algorithms provide stronger proportional fairness guarantees than other approaches, but are more difficult to implement and still require at least $O(\log N)$ time.

Motivated by the need for faster schedulers with good fairness guarantees, one of the authors developed Virtual-Time Round-Robin (VTRR) [17]. VTRR first introduced the simple idea of going round-robin through clients but

skipping some of them at different frequencies without having to reorder clients on each schedule. This is done by combining round-robin scheduling with a virtual time mechanism. GR^3 's intergroup scheduler builds on VTRR but uses weight ratios instead of virtual times to provide better fairness. Smoothed Round Robin (SRR) [9] uses a different mechanism for skipping clients using a Weight Matrix and Weight Spread Sequence (WSS) to run clients by simulating a binary counter. VTRR and SRR provide proportional sharing with $O(1)$ time complexity for selecting a client to run, though inserting and removing clients from the run queue incur higher overhead: $O(\log N)$ for VTRR and $O(k)$ for SRR, where $k = \log \phi_{\max}$ and ϕ_{\max} is the maximum client weight allowed. However, unlike GR^3 , both algorithms can suffer from large service time errors especially for skewed weight distributions. For example, we can show that the service error of SRR is worst-case $O(kN)$.

Grouping clients to reduce scheduling complexity has been used by [20], [8] and [23]. These fair queueing approaches group clients into buckets based on client virtual timestamps. With the exception of [23], which uses exponential grouping, the fairness of these virtual time bin sorting schemes depends on the granularity of the buckets and is adversely affected by skewed client weight distributions. On the other hand, GR^3 groups based on client weights, which are relatively static, and uses groups as schedulable entities in a two-level scheduling hierarchy.

The grouping strategy used in GR^3 was first introduced by two of the authors for uniprocessor scheduling [6] and generalized by three of the authors to network packet scheduling [4]. A similar grouping strategy was independently developed in Stratified Round Robin (StRR) [19] for

network packet scheduling. StRR distributes all clients with weights between 2^{-k} and $2^{-(k-1)}$ into class F_k (F here not to be confused with our frontlog). StRR splits time into scheduling slots and then makes sure to assign all the clients in class F_k one slot every scheduling interval, using a credit and deficit scheme within a class. This is also similar to GR^3 , with the key difference that a client can run for up to two consecutive time units, while in GR^3 , a client is allowed to run only once every time its group is selected regardless of its deficit.

StRR has weaker fairness guarantees and higher scheduling complexity than GR^3 . StRR assigns each client weight as a fraction of the total processing capacity of the system. This results in weaker fairness guarantees when the sum of these fractions is not close to the limit of 1. For example, if we have $N = 2^k + 1$ clients, one of weight 0.5 and the rest of weight $2^{-(k+2)}$ (total weight = 0.75), StRR will run the clients in such a way that after 2^{k+1} slots, the error of the large client is $\frac{-N}{3}$, such that this client will then run uninterruptedly for N tu to regain its due service. Client weights could be scaled to reduce this error, but with additional $O(N)$ complexity. StRR requires $O(g)$ worst-case time to determine the next class that should be selected, where g is the number of groups. Hardware support can hide this complexity assuming a small, predefined maximum number of groups [19], but running an StRR processor scheduler in software still requires $O(g)$ complexity.

GR^3 also differs from StRR and other deficit round-robin variants in its distribution of deficit. In DRR, SRR, and StRR, the variation in the deficit of all the clients affects the fairness in the system. To illustrate this, consider $N + 1$ clients, all having the same weight except the first one, whose weight is N times larger. If the deficit of all the clients except the first one is close to 1, the error of the first client will be about $\frac{N}{2} = O(N)$. Therefore, the deficit mechanism as employed in round-robin schemes doesn't allow for better than $O(N)$ error. In contrast, GR^3 ensures that a group consumes all the work assigned to it, so that the deficit is a tool used only in distributing work within a certain group, and not within the system. Thus, groups effectively isolate the impact of unfortunate distributions of deficit in the scheduler. This allows for the error bounds in GR^3 to depend only on the number of groups instead of the much larger number of clients.

A rigorous analysis on network packet scheduling [27] suggests that $O(N)$ delay bounds are unavoidable with packet scheduling algorithms of less than $O(\log N)$ time complexity. GR^3 's $O(g^2)$ error bound and $O(1)$ time complexity are consistent with this analysis, since delay and service error are not equivalent concepts. Thus, if adapted to packet scheduling, GR^3 would worst-case incur $O(N)$ delay while preserving an $O(g^2)$ service error.

Previous work in proportional share scheduling has focused on scheduling a single resource and little work has

been done in proportional share multiprocessor scheduling. WRR and fair-share multiprocessor schedulers have been developed, but have the fairness problems inherent in those approaches. The only multiprocessor fair queueing algorithm that has been proposed is Surplus Fair Scheduling (SFS) [7]. SFS also adapts a uniprocessor algorithm, SFQ, to multiple processors using a centralized run queue. No theoretical fairness bounds are provided. If a selected client is already running on another processor, it is removed from the run queue. This operation may introduce unfairness if used in low overhead, round-robin variant algorithms. In contrast, GR^3MP provides strong fairness bounds with lower scheduling overhead.

SFS introduced the notion of *feasible* clients along with a $O(P)$ -time weight readjustment algorithm, which requires however that the clients be sorted by their original weight. By using its grouping strategy, GR^3MP performs the same weight readjustment in $O(P)$ time without the need to order clients, thus avoiding SFS's $O(\log N)$ overhead per maintenance operation. The optimality of SFS's and our weight readjustment algorithms rests in preservation of ordering of clients by weight and of weight proportions among feasible clients, and not in minimal overall weight change, as [7] claims.

7 Conclusions and Future Work

We have designed, implemented, and evaluated Group Ratio Round-Robin scheduling in the Linux operating system. We prove that GR^3 is the first and only $O(1)$ uniprocessor and multiprocessor scheduling algorithm that guarantees a service error bound of less than $O(N)$ compared to an idealized processor sharing model, where N is the number of runnable clients. In spite of its low complexity, GR^3 offers better fairness than the $O(N)$ service error bounds of most fair queuing algorithms that need $O(\log N)$ time for their operation. GR^3 achieves these benefits due to its grouping strategy, ratio-based intergroup scheduling, and highly efficient intragroup round robin scheme with good fairness bounds. GR^3 introduces a novel frontlog mechanism and weight readjustment algorithm to schedule small-scale multiprocessor systems while preserving its good bounds on fairness and time complexity.

Our experiences with GR^3 show that it is simple to implement and easy to integrate into existing commodity operating systems. We have measured the performance of GR^3 using both simulations and kernel measurements of real system performance using a prototype Linux implementation. Our simulation results show that GR^3 can provide more than two orders of magnitude better proportional fairness behavior than other popular proportional share scheduling algorithms, including WRR, WFQ, SFQ, VRR, and SRR. Our experimental results using our GR^3 Linux implementation further demonstrate that GR^3 provides accurate proportional fairness behavior on real ap-

plications with much lower scheduling overhead than other Linux schedulers, especially for larger workloads.

While small-scale multiprocessors are the most widely available multiprocessor configurations today, the use of large-scale multiprocessor systems is growing given the benefits of server consolidation. Developing accurate, low-overhead proportional share schedulers that scale effectively to manage these large-scale multiprocessor systems remains an important area of future work.

Acknowledgements

Chris Vaill developed the scheduling simulator and helped implement the kernel instrumentation used for our experiments. This work was supported in part by NSF grants EIA-0071954 and DMI-9970063, an NSF CAREER Award, and an IBM SUR Award.

References

- [1] J. C. R. Bennett and H. Zhang. WF²Q: Worst-Case Fair Weighted Fair Queueing. In *Proceedings of IEEE INFOCOM '96*, pages 120–128, San Francisco, CA, Mar. 1996.
- [2] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, Sebastopol, CA, second edition, 2002.
- [3] R. Bryant and B. Hartner. Java Technology, Threads, and Scheduling in Linux. IBM developerWorks Library Paper. IBM Linux Technology Center, Jan. 2000.
- [4] B. Caprita, W. C. Chan, and J. Nieh. Group Round-Robin: Improving the Fairness and Complexity of Packet Scheduling. Technical Report CUCS-018-03, Columbia University, June 2003.
- [5] W. C. Chan. Group Ratio Round-Robin: An O(1) Proportional Share Scheduler. Master's thesis, Columbia University, June 2004.
- [6] W. C. Chan and J. Nieh. Group Ratio Round-Robin: An O(1) Proportional Share Scheduler. Technical Report CUCS-012-03, Columbia University, Apr. 2003.
- [7] A. Chandra, M. Adler, P. Goyal, and P. J. Shenoy. Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors. In *Proceedings of the 4th Symposium on Operating System Design & Implementation*, pages 45–58, San Diego, CA, Oct. 2000.
- [8] S. Y. Cheung and C. S. Pencea. BSFQ: Bin-Sort Fair Queueing. In *Proceedings of IEEE INFOCOM '02*, pages 1640–1649, New York, NY, June 2002.
- [9] G. Chuanxiong. SRR: An O(1) Time Complexity Packet Scheduler for Flows in Multi-Service Packet Networks. In *Proceedings of ACM SIGCOMM '01*, pages 211–222, San Diego, CA, Aug. 2001.
- [10] J. Cobb, M. Gouda, and A. El-Nahas. Time-Shift Scheduling – Fair Scheduling of Flows in High-Speed Networks. *IEEE/ACM Transactions on Networking*, pages 274–285, June 1998.
- [11] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM '89*, pages 1–12, Austin, TX, Sept. 1989.
- [12] R. Essick. An Event-Based Fair Share Scheduler. In *Proceedings of the Winter 1990 USENIX Conference*, pages 147–162, Berkeley, CA, Jan. 1990. USENIX.
- [13] P. Goyal, H. Vin, and H. Cheng. Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. *IEEE/ACM Transactions on Networking*, pages 690–704, Oct. 1997.
- [14] G. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, Oct. 1984.
- [15] J. Kay and P. Lauder. A Fair Share Scheduler. *Commun. ACM*, 31(1):44–55, 1988.
- [16] L. Kleinrock. *Computer Applications*, volume II of *Queueing Systems*. John Wiley & Sons, New York, NY, 1976.
- [17] J. Nieh, C. Vaill, and H. Zhong. Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 245–259, Berkeley, CA, June 2001. USENIX.
- [18] A. Parekh and R. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.
- [19] S. Ramabhadran and J. Pasquale. Stratified Round Robin: a Low Complexity Packet Scheduler with Bandwidth Fairness and Bounded Delay. In *Proceedings of ACM SIGCOMM '03*, pages 239–250, Karlsruhe, Germany, Aug. 2003.
- [20] J. Rexford, A. G. Greenberg, and F. Bonomi. Hardware-Efficient Fair Queueing Architectures for High-Speed Networks. In *Proceedings of IEEE INFOCOM '96*, pages 638–646, Mar. 1996.
- [21] R. Russell. Hackbench: A New Multiqueue Scheduler Benchmark. <http://www.lkml.org/archive/2001/12/11/19/index.html>, Dec. 2001. Message to Linux Kernel Mailing List.
- [22] M. Shreedhar and G. Varghese. Efficient Fair Queueing using Deficit Round Robin. In *Proceedings of ACM SIGCOMM '95*, pages 231–242, Sept. 1995.
- [23] D. Stephens, J. Bennett, and H. Zhang. Implementing Scheduling Algorithms in High Speed Networks. *IEEE JSAC Special Issue on High Performance Switches/Routers*, Sept. 1999.
- [24] D. Stiliadis and A. Varma. Efficient Fair Queueing Algorithms for Packet-Switched Network. *IEEE/ACM Transactions on Networking*, pages 175–185, Apr. 1998.
- [25] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288 – 289, Dec. 1996.
- [26] C. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Dept. of EECS, MIT, Sept. 1995.
- [27] J. Xu and R. J. Lipton. On Fundamental Tradeoffs between Delay Bounds and Computational Complexity in Packet Scheduling Algorithms. In *Proceedings of ACM SIGCOMM '02*, pages 279–292, Aug. 2002.
- [28] L. Zhang. Virtualclock: a New Traffic Control Algorithm for Packet-Switched Networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.

Short Papers

A Hierarchical Semantic Overlay Approach to P2P Similarity Search

Duc A. Tran
Computer Science Department
University of Dayton
Dayton, OH 45469
duc.tran@notes.udayton.edu

1 Introduction

One of the most important problems in information retrieval is similarity search. Informally, the problem is: given a similarity query, which can be a *point* query or a *range* query, we need to return a set of contents that are most relevant to the search criteria according to some semantic distance function. We propose EZSearch, a decentralized solution to this problem in the context of Peer-to-Peer (P2P) networks. EZSearch features the following for a network of N users. First, queries can be answered with $O(\log_k N)$ worst-case search time and low search overhead. Second, to maintain the hierarchy, a node keeps track of only $O(k)$ other nodes and failure recovery requires no more than $O(k)$ reconnections; these overheads are independent of the network size. Last but not least, the number of objects whose indices are stored at remote nodes is small and, therefore, so are the costs of index migration, storage, and validity.

2 Peer-to-Peer Similarity Search

We consider a P2P network where each node has a set of *data objects* to share with other nodes in the network. These data objects are described based on the vector space model used in information retrieval theory [1]. Each data object x is represented as a d -term semantic vector $T_x = (w_{1x}, w_{2x}, \dots, w_{dx})$, where each dimension t_i reflects the keyword, concept, or *term* associated with x and w_{ix} the weight to reflect the significance of t_i in representing the semantic of x . Without loss of generality, we assume that all the weight values belong to the interval $[0, 1]$.

We employ the commonly used Cosine distance function to measure the semantic similarity between two objects x and y : $\text{simdist}(x, y) = \cos^{-1} \frac{T_x \cdot T_y}{\|T_x\|_2 \|T_y\|_2}$ where $T_x \cdot T_y$ is the dot product between vectors T_x and T_y and $\|\cdot\|_2$ is the Euclidean vector norm. The smaller

$\text{simdist}(x, y)$ is, the more semantically similar are X and Y to each other.

We consider two types of queries: *point* queries and *range* queries. A point query is described by a term vector $Q = (w_{1Q}, w_{2Q}, \dots, w_{dQ})$. We expect to return those data objects x such that $\text{simdist}(x, Q)$ is minimum. In some applications, the user may also specify a small constant ϵ to find those objects such that $\text{simdist}(x, Q) < \epsilon$. There are two types of range query, namely *simple* and *composite*. A *simple* range query is described by a hyperrectangular region $Q = [\min_{1Q}, \max_{1Q}] \times [\min_{2Q}, \max_{2Q}] \times \dots \times [\min_{dQ}, \max_{dQ}]$. A *composite* range query is a set of simple range queries. For a range query Q , we expect to return those data objects that belong to the region Q . While the system is aimed to be fully decentralized, we assume that a new user knows at least one existing user before the former can join the network.

3 Proposed Solution: EZSearch

The basic idea behind EZSearch is as follows. Peers (i.e., user nodes) are partitioned into clusters. Each cluster contains nodes having similar contents and manages a subspace of indices (*peer_location* P , *term_vector* T_x), or an *index zone*. For a search, the simplest solution is to scan all the clusters, which, however, would incur a linear search time. Alternatively, similar to using search trees for logarithmic runtime search, we can build a decision hierarchical overlay on top of these clusters such that the search scope will be reduced by some factor if the query is forwarded from a layer of the hierarchy to a lower layer.

For building the cluster overlay, we propose to use the Zigzag hierarchy, which we originally devised for streaming multimedia [2, 3]. The main advantage of Zigzag is its capability to handle the dynamics of P2P networks. We first present Zigzag and then propose how similarity search can be fulfilled efficiently using this hierarchy.

3.1 Zigzag Hierarchy

Definition 1 [Zigzag hierarchy] A Zigzag- k hierarchy of N nodes is a multi-layer hierarchy of clusters recursively defined below: ($k > 3$ is a constant):

1. Layer 0 contains all peers.
2. If the number of peers at layer j is greater than $3k$, they are partitioned into clusters whose size is in $[k, 3k]$. Otherwise, we reach the highest layer, where peers form only a single cluster. The size of this highest-layer cluster is in $[2, 3k]$.
3. A layer- j cluster designates two member peers as its **head** and **associate-head**. The head automatically appears at layer $(j + 1)$. The cluster partition at layer $(j + 1)$ is the same as at layer j . An exception applies to the highest-layer cluster in which only the head role is needed but the associate-head role is not necessary.

An illustration is given in the top diagram of Figure 1, where 52 nodes are organized into a Zigzag-4 hierarchy. Hereafter, we denote by $head(\cdot)$ and $ahead(\cdot)$ the head and associate-head, respectively, of a cluster or a peer. Since a peer may have different associate-heads at different layers, we use the notation $ahead_i(P)$ to refer to the associate-head of P at layer i . For instance, in Figure 1, $ahead_0(22) = 21$, $ahead_1(22) = 17$. Below are the terms we use for the rest of the paper:

- **Foreign head:** A non-head non-associate-head cluster-mate Y of a peer X at layer $j > 0$ is called a “foreign head” of layer- $(j - 1)$ cluster-mates of X .
- **Super cluster:** A layer- j ($j > 0$) cluster is the super cluster of any layer- $(j - 1)$ cluster whose head appears in the layer- j cluster.

Definition 2 [Connectivity in Zigzag hierarchy] (illustrated by the top diagram of Figure 1)

- **Intra-cluster connectivity:** In a cluster, the associate-head has a link to every other non-head peer. E.g., in Figure 1 (top diagram), associate-head 17 of its layer-1 cluster has a link to all of its layer-1 non-head cluster-mates (peers 2, 5, 9, 13). An exception applies to the highest-layer cluster in which all peers have a link from its head (because there is no associate-head for this layer).
- **Inter-cluster connectivity:** The associate-head of a cluster has a link from one of its foreign heads. E.g., in Figure 1 (top diagram), associate-head 18 at layer 0 has a link from peer 13, which is one of peer 18’s

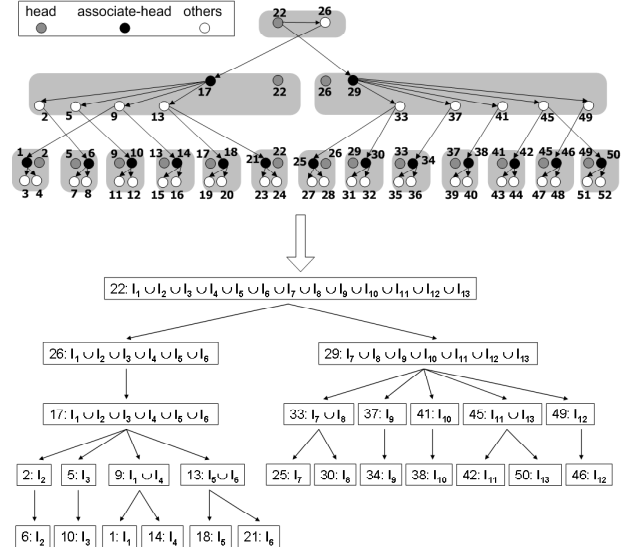


Figure 1: Top diagram: Connectivity in a zigzag-4 hierarchy of 52 nodes; Bottom diagram: Corresponding index zone assignments

foreign heads. *There is an exception: for a second-highest-layer cluster, if its associate-head does not have a foreign head, the associate-head has a link from the head of the highest-layer cluster.* For instance, associate-head 17 at layer 1 has a link from peer 26 which is the head of the highest-layer cluster.

The above rules guarantee a tree structure including all the peers; we call this tree the Zigzag tree. A Zigzag- k hierarchy of N peers provides the following desirable properties: (see [3] for complete proofs): (1) The maximum nodal degree in the Zigzag tree is $6k - 3$, (2) The maximum height of the Zigzag tree is $2\log_k N + 1$, (3) Recovery of a peer failure requires at most $6k - 2$ peer reconnections, and (4) As peers join and leave, a cluster may be split or merged with another cluster to satisfy the $[k, 3k]$ cluster size constraint. The worst-case number of peer reconnections due to a split or merger is $O(k)$.

3.2 Index Zone Assignment Policy

We propose to organize peers into a Zigzag hierarchy. Each cluster of this hierarchy is assigned a zone of the entire index space. Zone assignment is important to searching and follows the policy described below.

Definition 3 [Zone Assignment Policy]

1. At layer 0: Each layer-0 cluster owns a non-overlapped index zone, which is a set of hyperrectangles $\{[\alpha_1, \beta_1] \times [\alpha_2, \beta_2] \times \dots \times [\alpha_d, \beta_d]\}$, such that the union of all the zones of layer-0 clusters is

$I = [0, 1]^d$. This zone is known to both the head and associate-head of the cluster, and also said to be “covered by”, or “owned by”, the associate-head. The head will store the indices of those objects that belong to peers outside this cluster but lie inside its index zone.

2. At layer $j > 0$: Each peer P keeps a list of pairs $(child_i, zone(child_i))$ where $child_i$ is a child node of P in the Zigzag tree and $zone(child_i)$ the index zone covered by $child_i$. The index zone covered by peer P , denoted by $zone(P)$, is the union of these child zones. The index zone owned by a cluster is that covered by the associate-head of this cluster.

As an example, we consider the hierarchy in Figure 1 and suppose that the index zones owned by the 13 layer-0 clusters are I_1, I_2, \dots, I_{13} (respectively, from left to right). Thus, $zone(1) = I_1$, $zone(5) = I_2$, $zone(9) = I_3$, etc. Because peer 9 has two children (peer 1 and peer 14), peer 9 keeps the value $\{(1, I_1), (14, I_4)\}$ and $zone(9) = I_1 \cup I_4$. The index zone assignments are similar for other peers and shown in Figure 1 (bottom diagram). Since peers other than the heads and associate-heads at layer 0 do not own any index zone, they are not present in this index zone tree.

The advantage of the zone assignment policy is its support for efficient search. A search query just follows the links in the Zigzag tree to branches that lead to the smallest index zone(s) containing the query. The next subsection details the search algorithm.

3.3 Search Algorithms

We assume that peers are already organized into a Zigzag hierarchy and index zones are assigned to peers and clusters according to the zone assignment policy. We present here only the algorithm for range-query search. Algorithms for point queries can be generalized from this algorithm and can be found in [4]. Supposing that a peer P submits a range query Q , there are two scenarios:

Case 1: P is a leaf in the Zigzag tree (e.g., peers 15, 36, 40 in Figure 1) and P needs to process query Q . Since P does not have any index zone information, it sends the query to its associate-head $ahead_0(P)$. $ahead_0(P)$ computes $Q_1 = Q \cap zone(P)$. If $Q_1 \neq \emptyset$, some results of Q , that correspond to Q_1 , can be found locally. Indeed, $ahead_0(P)$ just needs to broadcast query Q_1 to all layer-0 clustermates asking them to return to peer P the objects inside Q_1 . Furthermore, when $head(P)$ receives Q_1 , if it stores any index (peer location P' , term vector T_x) such that $T_x \in Q_1$, $head(P)$ asks peer P' to send object x to peer P . We must also return the results that correspond to query $Q - Q_1$ if $Q - Q_1 \neq \emptyset$ because these results are not

in the local cluster. In this case, $ahead_0(P)$ creates a new query $Q_2 = Q - Q_1$. How $ahead_0(P)$ processes query Q_2 is similar to the case below.

Case 2: P is a non-leaf node in the Z-tree (e.g., peers 22, 37, 42 in Figure 1) and P needs to process query Q . In this case, P must own a zone $zone(P)$. Query Q is broken into two subqueries $Q_1 = Q \cap zone(P)$ and $Q_2 = Q - Q_1$, which will be handled in parallel as follows.

- Query Q_1 : If $Q_1 = \emptyset$, nothing needs to be done. Else, the results of Q_1 can be found in a layer-0 cluster reachable from peer P . By looking at the list $(child_i, zone(child_i))$ for every child, P breaks Q_1 further into subqueries Q_{11}, Q_{12}, \dots where $Q_{1i} = Q_1 \cap zone(child_i)$. (It is easy to prove that $Q_{1i} \cap Q_{1j} = \emptyset$ for every $i \neq j$.) The results for Q_{1i} can be found in a layer-0 cluster reachable from $child_i$. Hence, peer P just needs to forward these subqueries to the corresponding child peers. The handling of such a subquery at the corresponding child resembles that at peer P .
- Query Q_2 : If $Q_2 = \emptyset$, nothing needs to be done. Else, the results satisfying Q_2 cannot be found in any cluster reachable from P . In this case, P just needs to forwards Q_2 to the parent of P in the Zigzag tree. The handling of query Q_2 at this parent resembles the way P handles the original query Q .

Eventually, all the subqueries, created when necessary as above, will reach layer-0 clusters where the corresponding results can be found locally (like in Case 1). The collection of all these results is the final result for the original query Q initiated by peer P .

The search path length is at most the maximum distance in hops between two peers in the Zigzag tree, or $4 \log_k N + 2$. The search overhead is proportional to the total number of peers contacted for all the subqueries, which depends on the range of the original query. In our performance study, we found that this overhead is indeed very small.

3.4 Hierarchy Construction

Initially, there is only one peer in the network. It is the head of its self-formed cluster C , which grows larger as subsequent peers join. The index zone owned by this cluster is $I = [0, 1]^d$ and the ID of this zone is kept at the head node. When the cluster size exceeds $3k$, we need to partition C into two smaller clusters, C_0 and C_1 , whose sizes are in the interval $[k, 3k]$. We propose to partition I along a selected dimension t_l into two halves $I_{0l} = [0, 1]^{l-1} \times [0, 1/2] \times [0, 1]^{d-l}$ and $I_{1l} = [0, 1]^{l-1} \times [1/2, 1] \times [0, 1]^{d-l}$, each to be owned by C_0 and C_1 . It is possible that a peer in cluster C_0 has an object in I_{1l} (similarly, a peer in cluster C_1 may have an

object in I_{0l}). In this case, we store the index of this object in the other cluster. The number of such objects is called the index migration overhead. We want to minimize this overhead so that (1) the communication overhead due to index relocation is low, and (2) peers in the same cluster have highly similar objects. This is equivalent to minimizing $F = \sum_{P \in C_0} n_{1l}^P + \sum_{P \in C_1} n_{0l}^P$ where $n_{il}^P = \text{cardinality}(\{\text{object } x \in P \mid T_x \in I_{il}\})$. The algorithm for this purpose is run by $\text{head}(C)$ - the head of cluster C . Upon a request sent by $\text{head}(C)$, each peer P in C submits to $\text{head}(C)$ a set of tuples $(t_l, n_{1l}^P, n_{0l}^P)$, for all $l \in [1, d]$. Upon receipt of those sets from all the member peers, we can devise a simple greedy but optimal algorithm for $\text{head}(C)$ to find the best C_0, C_1 , and dimension t_l . The complexity of this algorithm is $O(dk \log_2 k)$.

Each cluster C_i randomly selects two nodes as its head $\text{head}(C_i)$ and associate-head $\text{ahead}(C_i)$ (the old head of cluster C , however, is preferred to remain head of the newly created cluster it belongs to). The heads will automatically belong to layer 1 and form a new cluster. Since layer 1 now is the highest layer, only the head needs to be designated; this head is randomly chosen between the two member peers. The index zone owned by this cluster is the union of the zones owned by its child clusters; in this case, it is $I_{0l} + I_{1l}$.

Having the Zigzag hierarchy initially constructed, we need maintain it under network dynamics such as when a peer publishes or removes objects, and joins or quits the network. The detailed solutions to these sub-problems are presented in [4], which shows that removal of a peer requires $O(k)$ peer reconnections, addition of a peer requires $O(\log_k N)$ peer contacts, and addition or removal of an object also requires $O(\log_k N)$ peer contacts.

4 Simulation Results and Future Work

We conducted simulation for EZSearch. Peers arrived at rate 2 peers per second and might quit the network randomly at anytime. Thus the contents and indices stored in the network changed dynamically. The results were promising. For instance, Figure 2 shows the effect of the constant k used in the Zigzag- k hierarchy. In all scenarios, the query and any of its subqueries do not travel more than 12 nodes (among 10,003 nodes) before knowing the locations of the answers. Normalized search overhead is computed as $\frac{n}{N \times V}$, where n is the number of nodes a query and its subqueries visit during the search, N the number of nodes currently in the system, and V the volume of the query. For a query of volume 0.2, the broadcast-based search would incur a normalized search overhead of $10003 / (0.2 \times 10003) = 5$. EZSearch has a normalized search overhead always less than 0.6 (on

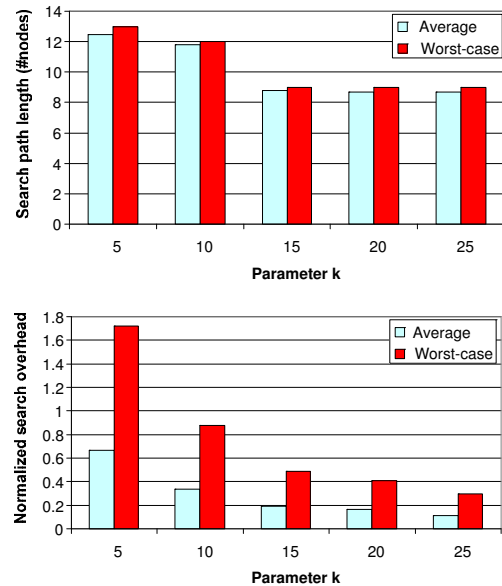


Figure 2: After the system runs for 5000 seconds, 10003 nodes are active. Each node has up to 10 2-d objects. 2000 queries are generated with ranges following a Zipf distribution in which about 80% of the queries have a volume of approximately 20% of the unit hypercube

average) and 1.8 (worst-case), and much smaller when k is larger. EZSearch therefore is fast and highly efficient. Our future work includes (1) refining the current algorithms for stronger index locality preservation within each cluster, and (2) considering various distributions of objects over peers.

References

- [1] M. Berry, Z. Drmac, and E. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.
- [2] D. A. Tran, K. A. Hua, and T. T. Do. Zigzag: An efficient peer-to-peer scheme for media streaming. In *IEEE INFOCOM*, San Francisco, CA, March-April 2003.
- [3] D. A. Tran, K. A. Hua, and T. T. Do. A peer-to-peer architecture for media streaming. *IEEE Journal on Selected Areas in Communications*, 22(1), January 2004.
- [4] D. A. Tran and H. Nguyen. EZSearch: Fast and scalable similarity search in peer-to-peer networks. Unpublished, January 2005.

A Parts-of-file File System

Yoann Padioleau and Olivier Ridoux

IRISA / University of Rennes 1
Campus universitaire de Beaulieu
35042 RENNES cedex, FRANCE

{padiolea,ridoux}@irisa.fr, <http://www.irisa.fr/lande>

Abstract

The *Parts-of-file File System* (PoIFS) allows read-write accesses to different *views* of a given file or set of files in order to help the user separate and manipulate different concerns. The set of files is considered as a mount point from which views can be selected as read-write files *via* directories. Paths are formulas mentioning properties of a desired view. Each directory contain a file (the view) which contains the parts of the mounted files that satisfy the properties. This service is offered generically at the file system level, and a plug-in interface permits that file formats, or application-specific details are handled by user-defined operators. Special plug-ins called *transducers* can be defined for automatically attaching properties to parts of files. Performances are encouraging; files of 100 000 lines are handled efficiently.

1 Introduction

A typical user working on a digital document performs alternatively one of the three following operations: *searching* for a desired piece of data, *understanding* a piece of data and possibly its relationship with other pieces of the document, and *updating* coherently related pieces. Examples of this situation can be found with textual documents, such as source programs or reports, text databases such as BibTeX files, agenda, or more recently, Web pages and XML documents. Hopefully, those documents have a structure that tools can exploit in order to help the user.

For searching, tools such as class browsers or hypertext tables of contents, and `grep` or the search button of an editor, provide navigation and querying methods. However, these tools suffer an important limitation in that they cannot be combined with each other to make search more efficient. For instance, a text file can be `grep`'ed for a searched string or navigated using a table of contents. However, there is no way to combine the `grep` program and the table of contents program so that one can get the smallest subset of the table of contents that covers the result of a `grep`.

For understanding a document, a commonly accepted practice is to build incomplete but simpler *views* of the document. A popular family of views is obtained by seeing the document at different depths. For instance, a table of contents offers a superficial view, but helps in understanding a document by giving a bird-eye's view on it. At a given depth, many views can also be defined. For instance, showing only the specifications of a program, hiding the debugging code, the comments, or showing only the functions sharing a given variable are possible views on a program file. Each such view helps in understanding one aspect of a program, and also helps in focusing on one task as the user is not visually bothered by irrelevant details. However, usual tools, such as a class browser or tools that support literate programming, provide only a few of these views, whereas all these views are conceptually simple to describe. The user's ability to express what a view should show and hide is often very limited.

For updating a document, views are also helpful. Indeed, an appropriate view can bring together related parts of a document that are distant in the original. For instance, gathering all conclusions of a book may help in updating them coherently. However, tools supporting views seldom support view update because it causes coherence problems between views, and between views and the original document.

The problem with all those tools is that they lack of shared general principles that would make it possible to incorporate new tools, supporting new kinds of navigation, query, views and updates, and that would make it possible to combine them fruitfully.

We can draw a parallel between the management of file contents and the management of file directories. Directories offer one rigid classification of files, in the same way as files offer one rigid organization of data. The possibility to associate several *properties* to a file and then to combine navigation and querying in *virtual directories* has been proposed in the past to help in the management of file directories [2, 3]. We propose in this article to associate several properties to *parts* of a file, and to consider views as *virtual files* built of selected parts to help in the management of file contents.

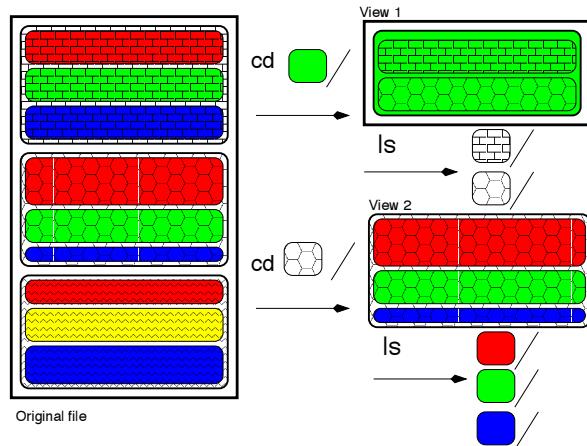


Figure 1: Symbolic representation of a file content

Figure 1 shows an iconic representation of the problem. The patterns represent the rigid structure of a file, and the grey levels represent different concerns. For instance, different patterns could correspond to different functions in a C program, and grey levels could correspond to heading, local declaration, body, or comments. As usual, concerns are scattered across structural boundaries. Hierarchical navigation must respect these boundaries. However, property-based navigation can focus on a concern across boundaries. Note that brick-pattern and dark-grey form an example of two properties that overlap. So, updating the brick-pattern view may also concern the dark-grey view.

Views must be first-class citizens, and they must be updated without restriction. We propose this new service at the file system level so that its impact is maximum without rewriting applications. So, several applications or users, with different requirements, can read and write the same file under their own visions. This new file system is called *Parts-of-file File System* (PofFS). Since criteria for defining a view are application specific, we propose that the file system operations only offer the generic background mechanisms, and that plug-ins could be defined to handle the application-level details.

2 Principles

We present PofFS as a shell-level demo because this is the simplest textual interface to a file system. However, more modern graphical interfaces like file browsers can also be used. We will say more on this subject at the end of the demo.

Consider a C program file, `foo.c`.

```
[1] % cat -n foo.c
1  int f(int x) {
2  int y;
3  assert(x > 1);
4  y = x;
5  fprintf(stderr, "x = %d", x);
```

properties line numbers	function:f	function:f2	var:x	var:y	var:z	debugging	specification
1	X		X				
2	X			X			
3	X		X				X
4	X		X	X			
5	X		X			X	
6	X			X			
7	X						
8		X			X		
9		X			X		
10		X					

Figure 2: A file context

```
6  return y * 2
7  }
8  int f2(int z) {
9  return z * 4
10 }
[2] % poffsmount foo.c /poffs
```

Command 1 displays the content of `foo.c`, and command 2 mounts `foo.c` on `/poffs`, using *transducers* for attaching properties to parts of file `foo.c`. The transducers are selected automatically using a mechanism similar to MIME types. This makes it easier to manage combinations of transducers without forgetting one. Properties are “this line belongs to the definition of *f*” (`function:f`), “this line mentions variable *x*” (`var:x`), “this line contains a trace instruction” (`debugging`), and “this line is an assertion” (`specification`). The attachment of properties to lines can be represented as a *lines* \times *properties* matrix which forms the *file context* (see Figure 2). In real-life, the *lines* \times *properties* matrix can be very large, e.g., 100 000 \times 10 000, but it is also very sparse, e.g., an average of 10 properties per line. Note that indexing is not local to parts. For instance, line 7 has property `function:f` because a declaration of function `f` has been found 6 lines above. A change at line 1 may affect properties of lines 2 to 7.

```
[3] % cd /poffs
[4] % ls
foo.c debugging/ specification/
function:f/ function:f2/ var:x/ var:y/ var:z/
```

Command 4 has two effects. First, it creates a *view* that contains all the parts of the file that correspond to this directory. As this directory is the mount point, the view has the same content as the original file. Second, it computes possible *refinements* to the current directory, and presents them to the user as sub-directories (`function:f/`, `debugging/`, ...).

```
[5] % cd function:f
```

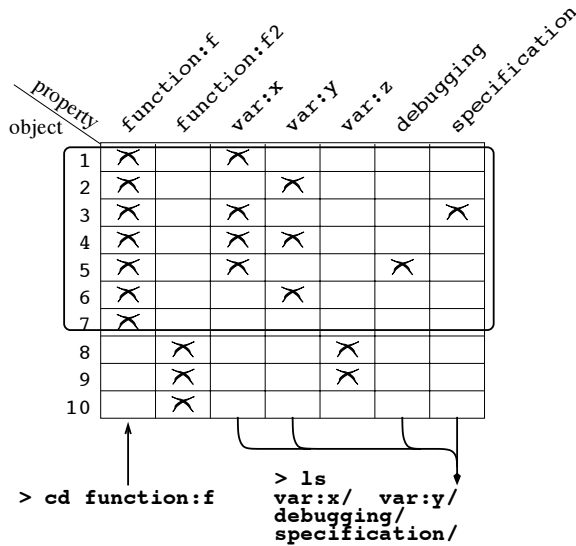



Figure 3: Navigation in a file context

```
[6] % ls
foo.c debugging/ specification/ var:x/ var:y/
```

Command 5 *refines* the current view by selecting parts of file that have property `function:f`. Command 6 shows how refinements are related to the current query. In directory `pofffs/function:f`, property `var:z` is no longer a refinement. This can be checked in the current view which contains only the code of function `f`, and contains no line related to variable `z` (see Figure 3). Moreover, `function:f/` is no longer a refinement, since it yields exactly the same view as the current one.

```
[7] % cd !(debugging|specification)
```

Command 7 illustrates the possibilities of the querying language. Negation is written `!`, and disjunction is written `|`. Character *slash* can be read as a conjunction. More sophisticated logics than propositional logic can be used by plugging *logic solvers* in the file system. For instance, program functions can be indexed by their types, and the types be compared using a type logic implemented as a pluggable module. So, *valued attributes* can be compared and filtered: e.g., `cd "type:?bool"` selects all functions with a boolean parameter. Similarly, `cd "function:^f.*"` selects all functions whose name starts with an 'f'. In this case a logic of strings (regex) is used. PoffFS also offers mechanisms for grouping resembling refinements; this reduces the size of answers to `ls`. For instance, sub-directories `function:f/` and `function:f2/` can be grouped in a directory `function:/`. Properties can also be grouped by the user in taxonomies, thus permitting to focus on a subset of the properties and making navigation easier.

```
[8] % ls
foo.c var:x/ var:y/
[9] % cat foo.c
```

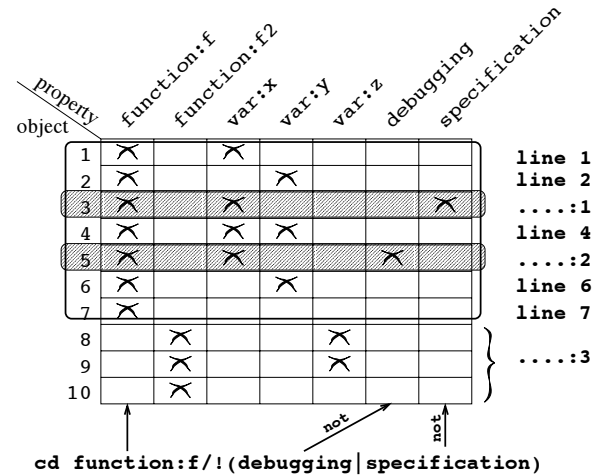


Figure 4: Creation of a view

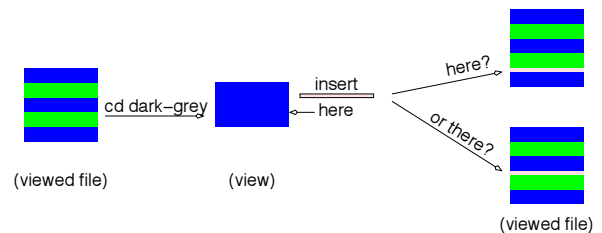


Figure 5: Why do we need marks of absence ?

```
int f(int x) {
int y;
.....:1
y = x;
.....:2
return y * 2
}
.....:3
```

Command 8 shows a list of sub-directories reduced to `var:x/` and `var:y/` (check on Figure 4). Command 9 displays the content of the current view. Groups of lines that do not satisfy the current query are replaced by *marks of absence*, e.g., `.....:1`. These marks will make it possible to back-propagate updates to the original file. Figure 5 illustrates the ambiguity of inserting a new piece in a file where missing parts are not marked.

```
[10] % cat foo.c | sed -e s/y/z > foo.c
```

Command 10 demonstrates that views can be *updated*, and can be so by any kind of tool. The effect of this command is to replace all occurrences of `y` by `z` *only in parts that belong to the current view*.

```
[11] % ls
foo.c var:x/ var:z/
```

Command 11 shows that updating a view affects property refinements (compare with results of command 8).

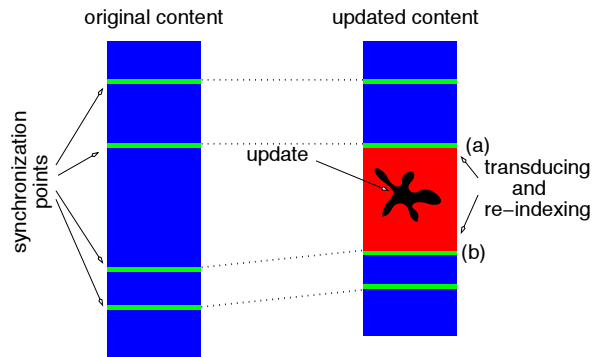


Figure 6: Re-indexing between synchronization points

```
[12] % pwd
/pofffs/function:f/!(debugging|specification)/
[13] % cd /pofffs
[14] % cat foo.c
int f(int x) {
  int z;
  assert(x > 1);
  z = x;
  fprintf(stderr, "x = %d", x);
  return z * 2
}
inf f2(int z) {
  return z * 4
}
```

Finally, command 14 shows that updating a view affects the other views. PoffFS maintains the coherence of all views by back-propagating view updates to the original file, and then to the other views. In the worst case, this could cause a complete re-indexing of the original file. However, it is often the case that a file can be split in several parts that do not depend on each other from the point of view of the properties attached by a transducer. We call *synchronization points* the boundaries of these independent parts. A sophisticated algorithm makes it possible to limit re-indexing to as few independent parts as possible (see Figure 6).

The shell interface must not be taken for the file system. What PoffFS actually offers is an implementation of operations `open`, `readdir`, `read`, `write`, etc. This makes it possible to adapt any existing interface that use these operations. For instance, PoffFS on-line demo uses an unmodified web file-browser. However, it is sometime better to devise a specialized interface. This is what we have done for an HTML agenda in which properties allow one to select different kinds of events. The web server always displays the current view of selected events simply because a URL contains a PoffFS path, and sub-directories are sorted according to their type (date, type of event, ...) and listed in menus.

3 Discussion and conclusion

The service of manipulating file contents is already offered by several kinds of application: e.g., text editors like Emacs, CIA [1], and IDEs (*Integrated Development Environments*) like Eclipse. These applications offer means for querying and navigation, and they also allow to hide parts of file. The novelty of PoffFS is to combine fruitfully query, navigation and view update at system level. We have used it in applications like text edition and programming, and also in trace and log analysis. In all cases, performances are encouraging.

The management of file contents at the system level has not evolved much since the first systems with stream files; files are units, querying and navigating work at the file level. Only read and write go inside files. Considering files as flat streams have been a fruitful abstraction to permit the combination of tools *via* pipes, redirection, etc. We have proposed to consider files as possible mount-points, to navigate in them, to extract views, and to update them. This raises the consistent view update problem between the views and the mounted file. We have proposed a mechanism of updatable views that solves this problem efficiently. The *Parts-of-file File System* (PoffFS) makes the structure of files virtual, and less tightly related to the physical model of a stream of characters. What PoffFS does is to recover structure in files and still permit the fruitful combination of tools.

Our current PoffFS prototype handles parts as sets of lines; further works will be to handle parts as sets of character positions. We have only used PoffFS with text files. However, nothing in principle prevents from using PoffFS with binary files (executable or multimedia) as long as a transducer exists. More experiments are required to assess the feasibility of using PoffFS with binary files, in particular with respect to real-time constraints of multimedia usage.

A prototype PoffFS and more information on this project can be down-loaded at the following URL:

<http://www.irisa.fr/LIS/PoffFS/>

This page makes also accessible a companion paper completing this article by describing the algorithms, benchmarks, and extensions to PoffFS. It presents also more precisely the principles and semantic of PoffFS, and gives a more complete account on the related works.

References

- [1] Y.-F. Chen, M. Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction system. *IEEE Transactions on Software Engineering*, 1990.
- [2] D.K. Gifford, P. Jouvelot, M.A. Sheldon, and J.W. O'Toole Jr. Semantic file systems. In *ACM Symp. Operating Systems Principles*, 1991.
- [3] Y. Padiou and O. Ridoux. A Logic File System. In *USENIX Annual Technical Conference*, 2003.

BINDER: An Extrusion-based Break-In Detector for Personal Computers

Weidong Cui, Randy H. Katz, and Wai-tian Tan

University of California, Berkeley and Hewlett-Packard Laboratories

{wdc,randy}@cs.berkeley.edu, dtan@hpl.hp.com

Abstract

Compromised computers have been a menace to both personal and business computing. In this paper, we tackle the problem of automated detection of break-ins of new unknown threats such as worms, spyware and adware on personal computers. We propose Break-IN DETectoR (BINDER), a host-based break-in detection system. Our key observation is that many break-ins make extrusions, stealthy malicious outgoing network connections. BINDER exploits a unique characteristic of personal computers, that most network activities are directly or indirectly triggered by user input. Since threats tend to run as background processes and thus do not receive any user input, the intuition behind BINDER is that only threats generate connections without user input. By correlating outgoing network connections and processing information with user activities, BINDER can capture extrusions and thus break-ins.

1 Introduction

A variety of threats such as worms, spyware and adware, have affected both personal and business computing significantly. Remotely controlled bot networks of compromised systems are growing quickly [12] and represent a menace to today's computing infrastructure. Many research efforts [8, 10, 14] and commercial products [13, 18] have focused on preventing break-ins by filtering known exploits or unknown ones exploiting known vulnerabilities. To protect computer systems from new threats, these solutions have two requirements. First, some central entities must rapidly generate signatures of new threats after they are detected. Second, distributed computer systems must download and apply these signatures to their local databases in time. However, these requirements can leave computer systems with obsolete signature database unprotected from newly emerging threats. In particular, worms can prop-

agate much more rapidly than humans can respond in terms of generation and distribution of signatures [11]. Instead of targeting at preventing infections, we focus on fast mechanisms for detecting break-ins after they occur that do not require a priori knowledge of exploit or vulnerability signatures. Such mechanisms can decrease the danger of information leak and protect computers and networks, and is complementary to existing prevention schemes.

Many threats send malicious outgoing network traffic that is usually happen unknown to users on the compromised personal computers. We refer to these *stealthy* malicious outgoing network activities as *extrusions*. The key feature of extrusions is that they are not triggered by user input. In contrast, most normal traffic in personal computers is initiated by users. Leveraging this anomaly of extrusions, we tackle the problem of automated detection of break-ins of new unknown threats such as worms, spyware and adware on personal computers in contrast to server computers. In this paper, we present Break-IN DETectoR (BINDER), a host-based system that detects break-ins on personal computers by capturing extrusions. To capture extrusions, BINDER correlates outgoing network connections (initiated by the local computer) and process information with user activities (key strokes and mouse clicks). BINDER can detect certain kinds of break-ins after they occur without a priori knowledge of exploit or vulnerability signatures.

2 Design Overview

The main objectives we want to achieve for the BINDER design are: (1) *minimal false alarms*: this is the critical requirement for any intrusion detection system to be useful in practice; (2) *generality*: BINDER should work for a large class of threats without the need for signatures beforehand; (3) *small overhead*: BINDER must *not* use intrusive probing and affect the performance of the computers it sits on; (4) *security with open design*: we want

to design BINDER so threats cannot bypass it by knowing its scheme.

Patterns of network traffic and system calls have been used for intrusion detection [4, 6, 17]. To the best of our knowledge, BINDER is the first system to take advantage of a unique characteristic of personal computers: user-driven activities. By trusting the user input, BINDER simply detects extrusions of break-ins by determining they are unrelated to user actions. In Section 4, we discuss how BINDER can verify a user input is not faked or tricked by break-ins.

A natural approach for BINDER to take is to correlate network traffic with user input directly. However, a “smart” threat can bypass it by monitoring user input and sending traffic at appropriate times. To avoid this, BINDER also uses process information to limit the correlation. The intuition behind it is that only malicious processes of break-ins generate connections without user input. BINDER assumes that the boundary between processes is protected by the operating system. In other words, break-ins cannot inject their malicious code into other running processes and must run as independent processes. Although this may not be guaranteed until the next generation operating systems [7] are available, a large class of today’s threats run as independent processes.

3 BINDER

BINDER detects break-ins by capturing extrusions. Instead of searching for conditions that can detect extrusions directly, BINDER looks for the cases where normal connections may be generated. This is in concert with our objectives of minimizing false alarms and detecting a large class of threats. By covering all normal cases, we can first control false alarms. Then, we can detect any threat that generates connections in a way that does not match any normal case.

3.1 Normal Connection Rules

In the following discussion, we use three kinds of events: user events (user input), process events (process start and process finish), and network events (connection request, data arrival and domain name lookup). A normal outgoing network connection of a process may be triggered either by user inputs directly (e.g., download a news web page after a mouse click) or indirectly (e.g., download an image file embedded in the news web page or refresh a news web page periodically after it is loaded) or by schedule (e.g., a system daemon or a software update check). These cases can be covered by three rules: (1) *intra-process* rule: a connection of a process may be triggered by a user input, data arrival or connection request

event of the same process; (2) *inter-process* rule: a connection of a process may be triggered by a user input or data arrival event of *another* process; (3) *whitelisting* rule: a connection of a process may be triggered according to a rule in the whitelist.

To verify if a connection is triggered by the *intra-process* rule, we just need to monitor all user and network activities of each single process. However, we need to monitor all possible communications among running processes to verify if a connection is triggered by the *inter-process* rule. This contradicts our objective of small overhead. Instead, we use the following two rules to approximate it: (1) *parent-process* rule: a connection of a process may be triggered by a user input or data arrival event received by its parent process before it is created; (2) *web-browser* rule: a connection of a web browser process may be triggered by a user input or data arrival event of other processes. The *web-browser* rule cannot be covered by the *parent-process* rule because, when a user clicks a hyperlink in a window of a process, the corresponding web page is loaded by an existing web browser process if there is any. The advantages of this approximation are that (1) the two rules do not require more information than the *intra-process* rule; (2) they cover a dominant fraction of connections triggered by the *inter-process* rule. The *whitelisting* rule can be classified into three categories: system daemons, software updates and network applications automatically started by the operating system.

3.2 Extrusion Detection Algorithm

The extrusion detection algorithm is based on the *intra-process*, *parent-process* and *web-browser* rule as well as whitelisting. It is actually about detecting normal connections correctly. If a connection is not triggered by any of the normal connection rules and thus is detected as anomalous, it is treated as an extrusion. The main idea of the extrusion detection algorithm is to limit the delay from a triggering event to a connection request event. There are three possible delays for a connection request though some of them may not exist. For a connection request made by process P , we define the three delays: (1) D_{new} : the delay since the last user input or data arrival event received by the parent process of P before P is created; (2) D_{old} : the delay since the last user input or data arrival event received by P (note that the triggering event can be from any process if P is a web browser process according to the *web browser* rule); (3) D_{prev} : the delay since the last connection request to the same host or IP address made by P . For normal connections, D_{old} and D_{new} are in the order of seconds while D_{prev} is in the order of minutes. Depending on how a normal connection is triggered, it must have at least one of the three

delays fall into a normal range.

The extrusion detection algorithm needs 3 parameters for the upper bound of the delays (defined as D_{new}^{upper} , D_{old}^{upper} , and D_{prev}^{upper}). We also assume BINDER can learn rules from previous false alarms. Each rule includes an application name (the image file name of a process) and a remote host name. The rule means any connection to the host made by a process of the application is always normal. Thus, given a connection request, it is a normal connection if it is in the rule set of previous false alarms, or is in the whitelist, or D_{prev} exists and is less than D_{prev}^{upper} , or D_{new} exists and is less than D_{new}^{upper} , or D_{old} exists and is less than D_{old}^{upper} . Otherwise, it is an extrusion.

3.3 Why BINDER Works

In this section, we discuss why connections made by a large class of threats can be detected as extrusions by the detection algorithm. When a threat breaks into a personal computer, the break-in can be split into two phases by the time of the first restart of the victim computer. The difference of these two phases is how malicious processes of a threat are started.

In the first phase, malicious processes are started either by an existing infected process or by a user. Connections made by those processes may not be initially detected as extrusions because, before they are started, their parent processes may have received user input (e.g., a user opens a virus attachment in an email client program) or data (e.g., a vulnerable process receives malicious traffic). However, BINDER can detect a break-in by observing even a single extrusion it makes. The chance of detecting a single extrusion is high unless an attacker crafts the exploit intentionally to evade BINDER in this phase.

In the second phase, malicious processes are started by the operating system without any user input or data arrival in history. Moreover, threats tend to run as background processes to avoid being detected or shutdown by computer users. A feature of background processes is that they do not receive any user input. BINDER can detect the break-in by capturing the first connection made by its malicious processes as an extrusion. This is because their parent processes do not receive any user input before they are created, and they do not receive any user input after it. So D_{old} , D_{new} and D_{prev} do not exist for such a connection. Therefore, BINDER can be guaranteed to detect break-ins of worms, spyware and adware after the victim computer is restarted.

4 Countermeasures and Solutions

When BINDER's scheme is known to attackers, there are potential countermeasures that allow break-ins to send

malicious traffic without being detected. Though we try to investigate all possible attacks against BINDER, we cannot argue that we have considered all possible vulnerabilities. To evade BINDER's detection, a break-in can (1) stop or remove BINDER from the compromised host; (2) fake user input by using system APIs or tricking a user to input on its window; (3) fake data arrivals by keeping receiving data from a collusive site on a connection triggered by a user; (4) fake whitelist applications by replacing their executables; (5) use covert channels to leak information. To eliminate these countermeasures, we can (1) run BINDER at kernel level and monitor its running status; (2) monitor related APIs and detect pop-up windows of processes that do not receive any user input; (3) add more constraints on how a normal connection may be triggered; (4) use the scheme proposed in [1]; (5) use techniques presented in [3].

5 Related Work

Many research efforts [8, 10, 14] and commercial products [13, 18] have focused on preventing break-ins by filtering known exploits or unknown ones exploiting known vulnerabilities. Anomaly-based intrusion detection [4, 6, 17] have been studied for detecting unknown exploits. The performance of anomaly-based approaches is very limited in practice due to its high false positive rate [2]. Computer worms and spyware [9] have been a menace to both personal computing and large networks. Fast worm detection and containment becomes critical since worms can propagate much more rapidly than human response [11]. Most research efforts [5, 15, 16] have focused on random scanning worms. Compared to previous work, BINDER has three features: (1) it detects break-ins without a priori knowledge of exploit or vulnerability signatures; (2) it leverages the unique characteristics of personal computers that most normal traffic is triggered by users to achieve minimal false positives; (3) it detects a large class of threats that infect personal computers.

6 Future Work

We are implementing a prototype of BINDER on Windows operating systems because they are the largest group attacked by the most threats [12]. We plan to evaluate BINDER in two environments. In a real world environment, we want to install BINDER on computers used for daily work and evaluate its performance on both false positives and false negatives. In a controlled testbed, we want to test BINDER with real world email worms to evaluate its performance on false negatives. We focus on email worms because it is harder for BINDER to detect

their break-ins due to the fact that they are usually triggered by user input. Our preliminary results show that BINDER can limit false alarms to once per week and detect break-ins of real world email worms and spyware successfully.

7 Conclusions

In this paper, we present the design of BINDER, a host-based system that detects break-ins of worms, spyware and adware on personal computers by capturing extrusions. BINDER takes advantage of a unique characteristic of personal computers: user-driven activities. By trusting the user input, BINDER simply detects break-in extrusions by determining they are unrelated to user actions. This implies a new direction for tackling the problem of intrusion detection on personal computers.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Erich Nahum, for their valuable comments and suggestions. We are grateful to Minghua Chen, Yanmei Li, Mukund Seshadri, Rui Xu, Fang Yu, Haibo Zeng, Jianhui Zhang and Wei Zheng for their generous help of allowing us to install and evaluate BINDER on their computers. We are thankful to Dan Ellis, Jaeyeon Jung, Jon Kuroda and Zhenmin Li for sharing virus emails with us. We would like to thank Chris Karlof, Yaping Li and Zhi-Li Zhang for their insightful comments on a draft of this paper. Special thanks go to Vern Paxson, David Wagner, Nicholas Weaver and Li Yin for their helpful discussion and valuable feedback.

References

- [1] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy. Digsig: Run-time authentication of binaries at kernel level. In *Proceedings of LISA*, November 2004.
- [2] S. Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Proceedings of ACM CCS*, November 1999.
- [3] K. Borders and A. Prakash. Web tap: Detecting covert web traffic. In *Proceedings of ACM CCS*, October 2004.
- [4] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
- [5] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *2004 IEEE Symposium on Security and Privacy*, May 2004.
- [6] W. Lee and S. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Transactions on Information and System Security*, 3(4), November 2000.
- [7] Microsoft, Next-Generation Secure Computing Base. <http://www.microsoft.com/resources/ngscb/default.aspx>.
- [8] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.
- [9] S. Saroiu, S. D. Gribble, and H. M. Levy. Measurement and analysis of spyware in a university environment. In *Proceedings of NSDI*, March 2004.
- [10] Snort, The Open Source Network Intrusion Detection System. <http://www.snort.org/>.
- [11] S. Staniford, V. Paxson, and N. Weaver. How to own the Internet in your spare time. In *Proceedings of the 11th Usenix Security Symposium*, August 2002.
- [12] Symantec Internet Security Threat Report. <http://enterprisesecurity.symantec.com/content.cfm?articleid=1539>, September 2004.
- [13] Symantec Norton Antivirus. <http://www.symantec.com/>.
- [14] H. J. Wang, C. Guo, D. R. Simon, and A. Zugmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM*, August 2004.
- [15] N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *Proceedings of the 13th Usenix Security Symposium*, August 2004.
- [16] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. Technical Report Technical Report HPL-2002-172, HP Labs Bristol, 2002.
- [17] Y. Zhang and V. Paxson. Detecting stepping stones. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.
- [18] ZoneAlarm. <http://www.zonelabs.com/>.

Proper: Privileged Operations in a Virtualised System Environment

Steve Muir, Larry Peterson, Marc Fiuczynski, Justin Cappos, John Hartman

Princeton University and the University of Arizona

{smuir, llp, mef}@cs.princeton.edu, {justin, jhh}@cs.arizona.edu

Abstract

Virtualised systems have experienced a resurgence in popularity in recent years, particularly in supporting a large number of independent services on a single host. This paper describes our work designing and implementing Proper, a service running on the PlanetLab system, that allows other services to perform privileged operations in a safe, controlled manner. We describe how implementing such a system in a traditional UNIX environment is non-trivial, and discuss the practical use of Proper.

1 Introduction

Operating systems face a fundamental tension between providing isolation and sharing among applications—they support the illusion that each application has the physical machine to itself, yet also allow those applications to share objects (files, pipes, etc.) with each other. General-purpose OSes typically provide a weak form of isolation (the process abstraction) with relatively unrestricted sharing between applications. In contrast, virtual machine monitors (VMMs) strive to provide strong performance isolation and privacy between virtual machines (VMs), and provide no more support for sharing between VMs than the network provides between physical machines.

The point on the design spectrum supported by any given system depends on the workload it is designed to support. General-purpose OSes run multiple applications on behalf of a single user, making it natural to favor sharing over isolation. Similarly, VMMs are often designed to allow a single machine to host multiple independent applications, possibly on behalf of independent organizations. In such a scenario, the applications have no need to share information and but require a predictable fraction of the physical machine's resources—hence, VMMs heavily favor isolation over sharing.

This paper investigates an alternative design point, motivated by the need for VMs to interact with each other in well-defined and controlled ways. All systems provide a means by which isolated components interact, but we are particularly interested in the problem of ‘unbundling’ the management of a set of VMs from the underlying VMM.

To enable multiple *management services*, it is necessary to ‘poke holes’ in the isolation barriers between VMs. These holes allow one VM to see and manipulate objects such as files and processes in another VM, providing a natural means for one VM to help manage another.

Toward this end, this paper describes Proper, a new ‘**Privileged Operations**’ mechanism that VMs can use to poke the holes that they need. Proper is straightforward to implement on a UNIX-based VMM such as that used in PlanetLab, and enables useful management services that run today on PlanetLab.

2 The PlanetLab Virtualised Environment

Our work is conducted in the context of PlanetLab, a global network of 500+ PCs used by researchers at over 250 institutions to develop new network services and conduct network-based experiments. Each PlanetLab node runs a modified Linux kernel, which provides a virtualised Linux environment for each user, with 30–40 users active on each node at any given time. Although Proper was originally developed as a component of the PlanetLab node infrastructure we believe that it is also applicable to other virtualised systems e.g., VMWare, Xen, Denali.

PlanetLab provides each user with one or more *slices*—a set of resources bound to a virtual Linux environment and associated with some number of PlanetLab nodes. Each node runs a Linux kernel modified with two packages: Vservers [3] supports multiple Linux VMs running on a single kernel instance, while CKRM provides a resource management framework. Each PlanetLab slice is instantiated as a combination of a Vserver, providing *namespace isolation*, and a CKRM class, providing *performance isolation*.

Namespace isolation is the property of a virtualised system where each VM executes in its own namespace, and is required so that each VM can configure its own namespace without interference with or by other VMs e.g., a VM should be able to install packages in its own filesystem without concern as to whether another VM requires a different version. Performance isolation between two VMs allows each VM to consume and manage resources e.g., CPU, physical memory, network bandwidth,

Operation	Description
<code>open_file(name, options)</code>	Open a restricted file
<code>set_file_flags(name, flags)</code>	Change file flags
<code>get_file_flags(name, flags)</code>	Get file flags
<code>mount_dir(source, target, options)</code>	Mount one directory onto another
<code>unmount(source)</code>	Unmount a previously-mounted directory
<code>exec(context, cmd, args)</code>	Execute a command in the context (slice) given
<code>wait(childid)</code>	Wait for an executed command to terminate
<code>create_socket(type, address)</code>	Create a restricted socket
<code>bind_socket(socket, address)</code>	Bind a restricted socket

Table 1: Operations supported by Proper 0.3

without affecting other VMs.

Although namespace isolation is essential in order to support multiple VMs on a single physical system, it presents a barrier to cooperation between those VMs. Furthermore, it also prevents VMs from getting full visibility into the details of the host system, thus limiting monitoring and system management. Hence the need to provide a means for selected VMs to break their namespace isolation in carefully controlled ways.

3 Architecture and Implementation

Proper provides unprivileged slices with access to privileged operations in a controlled manner. Here we present the client-visible architecture of Proper, which we believe to be equally applicable to any virtualised system, such as Xen [1], Denali [8], etc., not just the PlanetLab environment.

3.1 The Proper Client API

Proper enables two different types of ‘*VM escapes*’: inter-VM communication, and access to resources outside the VM e.g., VMM internals, such as VM configuration. While access to VMM internals requires cooperation by the VMM, certain types of inter-VM communication may be possible without a service like Proper. For example, most VMMs permit filesystems between VMs using NFS or SMB. However, the VMM may be able to support direct filesystem sharing in more flexible and/or higher-performing ways, so Proper provides a high-level filesystem-sharing operation using a VMM-specific implementation.

The operations supported by Proper are chosen according to the following principles:

1. Client interactions with Proper should be minimised—once an initial request has been completed subsequent operations should only require interactions with the host VM.
2. Proper operations should be compatible with equivalent intra-VM operations—opening a file through Proper should yield a standard file descriptor.

3. The API should be as general and flexible as possible—there should be a single `open` operation rather than separate operations to read and write files, pipes and devices.

It is important to minimise the overhead due to communication between the client application and the Proper service—a single request grants access to a restricted object, and subsequent requests use the host VM’s standard data operations. For example, when opening a file it is acceptable to require the initial ‘`open`’ request be sent to Proper, but forcing every read/write request to also use Proper would impose a performance penalty. We discuss such effects later in Section 4.2.

Table 1 shows the current Proper API, with each group of operations briefly discussed in the following section. The API is primarily driven by the requirements of PlanetLab users, and is thus not exhaustive, but gives a good idea of the type of operations we envision being supported.

3.2 Implementing Proper

As well as describing our implementation of Proper, we present a strawman implementation of each operation group that illustrates how one might implement Proper on another virtualised system. We assume that Proper exists as a component of the VMM ‘root’ context that receives requests from clients running in a VM; alternatives are possible, such as modifying the OS inside each VM to be ‘Proper-aware’, but we believe that such modifications are unnecessary.

3.2.1 File Operations

The `open_file` operation allows an application to access a file outside its own VM, while the `get_` and `set_file_flags` operations support manipulation of restricted file flags i.e., flags that may be used by the VMM to support copy-on-write sharing of files between VMs, and hence must not be modifiable within the VM.

Opening a file can be supported in UNIX-like VMs by exploiting the similarities between file descriptors and network sockets: Proper opens the restricted file and

passes the data to the client using a network socket. Unfortunately, while the client can readily use a socket in place of a file descriptor for reading and writing data, other operations may reveal that the ‘file’ is in fact a socket. In PlanetLab we exploit the fact that each VM runs atop the same kernel as Proper to achieve our transparency goal: Proper passes the opened file descriptor, which is indistinguishable from that obtained if the client opened the file directly, to the client using a UNIX domain socket.

3.2.2 Directory Operations

A common requirement in virtualised systems is that two VMs share parts of their filesystem e.g., when one wishes to manage the other’s filesystem. As stated earlier, sharing can often be accomplished directly between two VMs using a protocol such as NFS, but the VMM may also support a direct sharing mechanism. For example, in PlanetLab all VMs exist as disjoint subtrees of a single filesystem, so one can take advantage of Linux’s ‘bind-mount’ facility to graft (bind) one subtree onto another. This feature is used by the Stork configuration manager (see Section 4.1) to manage client VMs.

3.2.3 Process Execution

A client application may wish to create processes outside of its own VM: to perform some privileged task in the ‘root’ context e.g., create a new VM, or to act as, say, a remote access service for another VM by creating processes in that VM in response to authenticated network requests. Both cases may require that long-running processes are created e.g., a remote login shell, so the `exec` operation is actually performed asynchronously, and client use a `wait` operation to wait until the process terminates. The client passes file descriptors for use as standard I/O so Proper need not be involved in data passing between the child process and the client; alternatively, sockets could be used as in the generic `open_file` implementation described above.

3.2.4 Network Socket Manipulation

The final class of operations supported by Proper allow clients within a VM to request privileged access to the VMM’s network traffic. For example, a VMM may multiplex a single network interface onto multiple per-VM interfaces, with each VM only being able to ‘see’ its own traffic. A traffic monitoring or auditing program, running in an unprivileged VM, wishes to receive a copy of every packet sent or received on the physical interface, so uses Proper to create a privileged ‘raw’ socket—Proper verifies that the auditing service is authorised to do so before configuring the network subsystem to dispatch copies of every packet to the raw socket.

Similarly, an application may wish to bind to a specific network port in order to receive all packets sent to that

port on this machine—since port space is often shared by all VMs using a physical network interface this requires coordination with the VMM. In PlanetLab we allow slices to bind ports above 1024 in a first-come, first-served fashion, but require use of Proper to bind restricted ports below 1024, thus ensuring that only the specific slice authorised to bind a particular port can do so.

4 Experiences Using Proper on PlanetLab

Proper has been deployed and in-use on PlanetLab for about 6 months, supporting a number of services. These include both core infrastructure e.g., network traffic auditing, and user services. Most importantly, Proper has enabled elimination of ‘privileged’ slices that were used in the previous version of the PlanetLab system to perform many of those functions.

4.1 An Example Service: Stork

Stork is a PlanetLab service that provides package management functionality to other services. It allows users to associate a set of packages with their slice and takes care of downloading and installing the software into the slices and keeping it up-to-date. Stork allows package contents to be shared between slices, reducing the software footprint on a PlanetLab node.

Stork is responsible for downloading packages from a package repository, maintaining a per-node package cache, and installing packages in client slices. When a package is installed in a client slice the contents must be transferred from Stork to the client. One way to do this is over a socket, but this is inefficient and prevents sharing of files between slices (see below). Instead, Stork mounts the appropriate package directory read-only into the client’s filesystem using `mount_dir`. This gives the client access to the files in the package directory without being able to modify the directory structure.

Of course, many slices may install the same packages, making it desirable to share the package contents between the slices. However, modifications made by one slice should not be visible to any other. Ideally, each slice should have a copy-on-write version of each file; unfortunately, this functionality is not available in PlanetLab. Instead, Stork relies on sharing files read-only between slices. Files that may be modified are not shared; typically these are a few configuration files for each package. Most files are shared, dramatically reducing the amount of disk space required to install packages in slices.

Stork makes shared files read-only using the Proper `set_fileflags` operation to set the `NOCHANGE` flag on the files when it unpacks them. When unpacking a package the Stork client creates hard links to the read-only files to put the files in the proper locations inside the client filesystem; writable files are copied instead of linked. The installation scripts are then run in the client

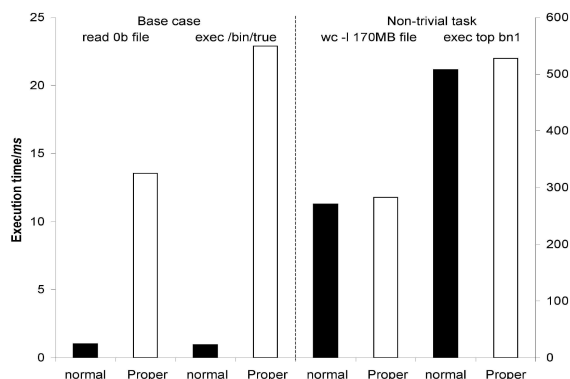


Figure 1: Overhead of Proper for various tasks

slice, and the package directory is unmounted. At this point the client slice has either hard links to or copies of the files in Stork's file system. The client is prevented from modifying shared (linked) files by the NOCHANGE flag, but can remove (unlink) the file to perform a private replacement.

4.2 Evaluation of Proper Overhead

One possible concern with our implementation of Proper is that invoking privileged operations using RPC may impose significant overhead. To address this we measured the overhead for a couple of operations, `open_file` and `exec`, for both trivial base cases and more realistic tasks—reading a large file and running a complex program respectively.

Figure 1 shows these results (measured on an idle 3GHz Pentium 4 with 1.25GB RAM): while the base case overhead, shown in the left half, is about 12–22ms, this is negligible for the non-trivial tasks, as shown on the right side of the graph. This overhead consists of two components: a client-side RPC overhead, typically 7.5ms, which is operation-independent; and an operation-dependent server-side latency of an additional 5–15ms.

5 Related Work

Much existing work on virtualisation techniques has some degree of relevance to Proper. Systems such as *VMWare* [7], *Xen* [1], *Denali* [8], *Zap* [4] and *Solaris Zones* [6] all provide users with virtualised environments, and often utilise a system-specific method for providing direct access from the VM to the VMM. For example, *VMWare* allows users to install extensions to popular 'guest' OSes, such as Windows, that permit direct access to host files in a similar manner to the bind mounts facilitated by Proper, while *Xen* allows multiple VMs to access physical devices through a special 'privileged' VM that runs unmodified Linux drivers.

Zap and *Zones* are perhaps closest to PlanetLab since

the environment they support is essentially the same as that provided by *Vservers* i.e. a thin layer on top of a UNIX-like kernel. Since each addresses a slightly different problem from PlanetLab it should be illustrative to consider what facilities a service like Proper should provide in those systems.

Another area of related work is in the security community, where researchers have investigated schemes for *system call interposition* e.g., *Ostia* [2], *Systrace* [5] in order to allow administrators to restrict the ability of unmodified applications to execute certain system calls. These systems adopt many of the same implementation solutions as our PlanetLab implementation of Proper and could thus be leveraged to provide an framework for integrating Proper with unmodified clients.

6 Conclusions

As part of the PlanetLab project we found that it was necessary to give unprivileged clients running inside a virtual machine access to certain privileged operations. This was accomplished with Proper, a user-level service running in the privileged root VM that performs operations on behalf of those unprivileged clients.

The key insight from this work is that supporting communication between VMs requires a degree of support from the underlying virtual machine monitor: virtualisation at the system call level readily supports certain forms of inter-VM communication, whereas more thoroughly virtualised systems are likely to require some modification to support the required forms of sharing and communication.

References

- [1] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proc. 19th SOSP* (Lake George, NY, Oct 2003).
- [2] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proc. 2004 Symposium on Network and Distributed System Security* (2004).
- [3] LINUX VServers PROJECT. <http://linux-vserver.org/>.
- [4] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proc. 5th OSDI* (Boston, MA, Dec 2002), pp. 361–376.
- [5] PROVOS, N. Improving Host Security with System Call Policies. In *Proc. 12th USENIX Security Symposium* (Washington, DC, Aug 2003), pp. 257–272.
- [6] TUCKER, A., AND COMAY, D. Solaris Zones: Operating System Support for Server Consolidation. In *3rd Virtual Machine Research and Technology Symposium Works-in-Progress* (San Jose, CA, May 2004).
- [7] VMWare. <http://www.vmware.com/>.
- [8] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and Performance in the Denali Isolation Kernel. In *Proc. 5th OSDI* (Boston, MA, December 2002), pp. 195–209.

AMP: Program Context Specific Buffer Caching

Feng Zhou, Rob von Behren and Eric Brewer

Computer Science Division
University of California, Berkeley
{zf,jrvb,brewer}@cs.berkeley.edu

Abstract

We present Adaptive Multi-Policy disk caching (AMP), which uses multiple caching policies within one application, and adapts both which policies to use and their relative fraction of the cache, based on program-context specific information. AMP differentiates disk requests based on the program contexts, or code locations, that issue them. Compared to recent work, AMP is unique in that it employs a new robust scheme for detecting looping patterns in access streams, as well as a low-overhead randomized way of managing many cache partitions. We show that AMP outperforms non-detection-based caching algorithms on a variety of workloads by up to 50% in miss rate reduction. Compared to other detection-based schemes, we show that AMP detects access patterns more accurately for a series of synthesized workloads, and incurs up to 15% fewer misses for one application trace.

1 Detection (classification) Based Caching

Modern applications access increasing amounts of disk data and have widely varying access patterns. These access patterns deviate from traditional OS workloads with temporal locality. Recent work on *detection* or *classification* based caching schemes [2, 5, 4] exploits consistent access patterns in applications by explicitly detecting the patterns and applying different policies. In this paper, we present Adaptive Multi-Policy caching, a detection-based caching system with several novel features.

One major aspect of the design of a detection-based caching scheme is the *pattern detection algorithm*. UBM [5] detects loops by looking for accesses to physically consecutive blocks in files. One obvious problem is that some loops are to blocks not physically consecutive. DEAR [2] detects loops and probabilistic patterns by sorting blocks by last-access time and frequency. Although it detects non-consecutive loops, it's more expensive and brittle against fluctuations in block ordering. PCC [4] resorts to a simple counter-based pattern detection algorithm. Essentially a stream with many repeating accesses is classified as looping. This scheme, although simple, risks classifying temporally clustered streams with high locality as loops.

AMP features a new access pattern detection algorithm. It detects consecutive and non-consecutive loops and is robust against small reorderings. Its overhead is small and can be made constant per access, independent of working set or memory size.

AMP is also novel in the way it manages the cache parti-

tions. Both UBM and PCC evict the block with the least estimated “marginal gain”. Because the marginal gain estimation changes over time, finding this block can be expensive. AMP, in contrast, uses a randomized eviction policy that is much cheaper and robustly achieves similar effectiveness.

One decision differentiating these approaches is the *definition of an I/O stream* to do detection on. For example, UBM [5] does per-file detection and DEAR is based on per-process detection. AMP and PCC [4] do per-program context (referred to as program counter in [4]) detection. The basic idea is to separate access streams by the *program context* (identified by the function call stack) when the I/O access is made, with the assumption that a single program context is likely to access disk files with the same pattern in the future. This PC-based approach differs radically from previous methods and is the key idea in both PCC and AMP, although the two systems were developed concurrently and independently. Interested readers are referred to [4] and [7] for further motivations for this approach and application traces showing its effectiveness.

2 Design

AMP is composed of two components. The first component, *Access Pattern Detector*, uses an access-pattern detection algorithm to determine the access pattern for each program context issuing I/O calls. The other component, *AMP Cache Manager*, subsumes the original OS cache manager. It maintains a default cache partition that holds all “normal” blocks using a default policy such as ARC or LRU (assumed to be ARC later on). Then, it maintains one partition for each “optimized” program context, using the appropriate policy for that program context. It continuously adapts to the workload and adjusts the sizes of the partitions.

2.1 Access-Pattern Detection

The AMP access pattern detector assigns one of the following block access patterns to each program context, similar to UBM and PCC: **One-shot** for one-time accesses; **Looping** for repeated accesses to a series of blocks in the same or roughly the same order, either physically consecutive or not. **Temporally clustered** [1] for accesses characterized by the property that blocks accessed recently are more likely to be accessed in the future. **Others** when none of the above apply.

PCs that always issue one-shot accesses are easy to identify, by simply observing if no blocks accessed by it are

accessed again. The other patterns are distinguished based on a metric we call *average reference recency*. The intuition is that if a sequence is temporally clustered, then the more recently a block was accessed, the more likely it is to be accessed again. The exact contrary holds for looping sequences. Hence, one way to distinguish these access patterns is to measure the *recency* of blocks accessed. Concretely, we measure this *reference recency* of an access by looking at the *relative position* of the previous appearance of the same block in the list of all previously accessed blocks, ordered by their last reference time.

Formally, for the i -th access in a sequence, we let L_i be the list of all previously accessed blocks, ordered from the oldest to the most recent by their last access time. Note that each block only appears in L_i at most once, at its position of last access. Thus if the access string is [4 2 3 1 2 3], with time increasing to the right, we have $L_6 = \{4, 3, 1, 2\}$, and $L_7 = \{4, 1, 2, 3\}$, although we don't yet know the seventh block. Let the length of the list be $|L_i|$, and the position of the block of interest be p_i , with the oldest position being 0 and newest position being $|L_i| - 1$.

Define the *reference recency* R_i of the i -th access as:

$$R_i = \begin{cases} p_i/(|L_i| - 1), & |L_i| > 1 \\ 0.5, & |L_i| = 1 \\ \perp, & \text{undefined for first access} \end{cases}$$

Then the *average reference recency* \bar{R} of the whole string is simply the average of all defined R_i .

Example 1. Consider looping access string [1 2 3 1 2 3]. For the second access to block 1, $i = 4$ and $L_4 = \{1 2 3\}$. Thus $p_4 = 0$, $R_4 = \frac{0}{3-1} = 0$. Also $L_5 = \{2, 3, 1\}$ and $p_5 = 0$, and thus $R_5 = 0$. Similarly, $R_6 = 0$ too, and in fact any pure looping pattern will have $R_i = 0$ and thus $\bar{R} = 0$. **Example 2.** Consider temporally clustered string [1 2 3 4 4 3 4 5 6 5 6]. With the calculations omitted, we get $\bar{R} = 0.79$.

In general, for pure loop sequences such as example 1, $\bar{R}=0$. For highly temporally clustered sequences \bar{R} is close to 1. It is also easy to see a uniformly random access sequence has $\bar{R} = 0.5$. In this sense, the *average reference recency* metric provides a measure of the correlation between recency and future accesses. If $\bar{R} > 0.5$, recently accessed blocks are more likely than average to be accessed in the near future, vice versa.

The \bar{R} values can be estimated either continuously, updating results as each I/O request is issued, or periodically, in a record-then-compute fashion. A continuous detector using exponential moving average of R values as \bar{R} instead of the definition above can respond faster to changes in workload. In contrast, the periodical one can be easier to implement, because it could be done at user-level and needs less interaction with the cache manager.

The pattern detector categorizes all contexts with $\bar{R} < T$ as having looping patterns, where T is an adjustable threshold. We found $T = 0.4$ to be a good value in experiments. The \bar{R} metric is quite robust against small permutations

of accesses. For example, the relative position of a block changes very little if access to it is exchanged with the access before or after it.

Block sampling. It is easy to see that the cost of computing \bar{R} per access is $O(|L|)$. This calculation could hence become rather expensive for PCs accessing a large number of blocks. Sampling can be applied to reduce this cost. A quick calculation reveals that by sampling $1/m$ pages, the total overhead could be reduced to roughly $1/m^2$ of the original. We could also adapt the sampling rate such that $|L|$ is bounded, thus limits the per-access overhead to a constant upper-bound. For details, see [7].

2.2 Low-overhead Partitioned Cache Management

The *Cache Manager* manages the cache according to the access pattern of each program context. AMP bases cache partition sizing decisions marginal gain estimation [5]. However, it differs from previous work in the way marginal gains are used. Both UBM and PCC evict from the partition with the smallest estimated marginal gain when a free block is needed. Unfortunately, finding this block can be expensive. Moreover, gain estimations are often inaccurate and delayed, which may lead to a large number of wrong evictions and overcorrections. AMP avoids these problems by using a *randomized* eviction policy and allowing cache partitions compete for new blocks. When a cache miss occurs, AMP forces some *other* randomly chosen partition to evict a page to free memory for the new page. This serves to increase the size of each partition proportional to its marginal gain; over time the partition sizes move towards a balance of equal marginal gain. This achieves the same local optimum as previous approaches but with far lower overhead.

The actual adaptation algorithm is briefly as follows (See [7] for complete specification and justification). When a cache miss happens, a free block is allocated if one is available. Otherwise, an occupied block needs to be evicted. If the missed block is in the ghost queues of ARC ($B1$ and $B2$ in [6]), a block from a random MRU partition is evicted. If instead the access is from a looping context, we evict from a random partition with probability $arc_ghost_len/loop_size_i$, where arc_ghost_len is the length of the ghost queues of the ARC partition and $loop_size_i$ is the estimated number of blocks in every loop of the missed MRU partition. Otherwise, we just evict the MRU block of the missed partition.

2.3 Linux Implementation

We have implemented AMP for Linux 2.6.8.1. The program contexts are identified by walking the user-level stack and hashing together function return addresses. The AMP cache manager is implemented by extending the Linux buffer cache, called *page cache*. The fact that buffer caching is tightly integrated into the virtual memory system in Linux poses some challenge to the implementation, as discussed in [7]. The pattern detector is implemented at user-level and operates periodically. It calls upon a kernel trace collector periodically to collect sampled disk

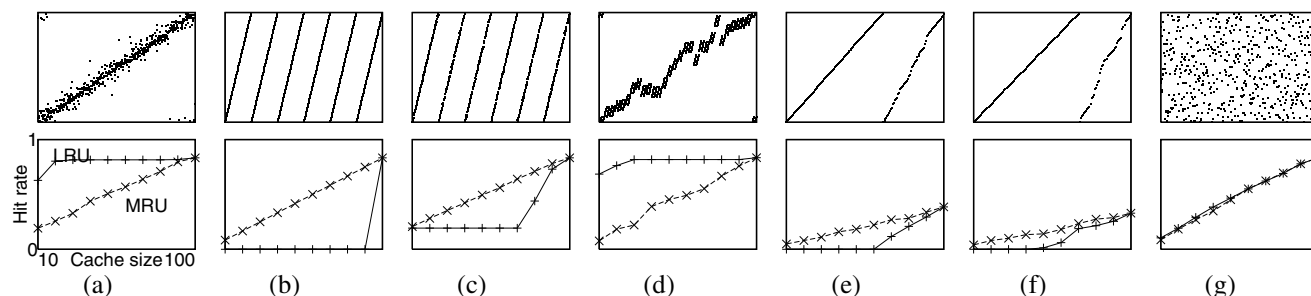


Figure 1: Traces used to compare detection schemes and their hit rates with LRU and MRU

Stream	AMP (R)	DEAR	PCC
a	other (0.755)	other	<u>loop</u>
b	loop (0.001)	loop	loop
c	loop (0.347)	<u>other</u>	loop
d	other (0.617)	other	<u>loop</u>
e	loop (0.008)	loop	loop
f	loop (0.010)	<u>other</u>	<u>other</u>
g	other (0.513)	other	<u>loop</u>

Table 1: Access pattern detection results of streams in Figure 1. Incorrect results are underlined.

I/O trace, along with corresponding program context information. The detection results are fed back to the kernel using the `/proc` file system interface.

3 Evaluation

3.1 Detection Scheme

We compared the AMP access detection scheme to DEAR[2] and PCC[4] using simulation. We implemented these based on the specifications in [2] and [4], respectively. All schemes work well for detecting pure looping patterns. Therefore we focus our experiments on accesses with patterns but, more importantly, irregularities. We synthesize several such access streams, as shown in Figure 1. Each stream accesses blocks numbered from 1-100. (a) shows temporally clustered accesses slowly moving through a file. (b) is pure loops and (c) is loops with each block moved around randomly (expected distance 0.5). (d) is temporally clustered with local loops. (e) and (f) are loops in which a block is accessed again with probability 0.6 and 0.5 respectively. (g) is uniformly random. The MRU and LRU hit rates in Figure 1 clearly indicate the best caching policy for each stream. Detection results of the algorithms are shown in Table 1.

Table 1 shows that AMP detects the correct pattern each time. DEAR detects the correct patterns except for (c) and (f). The DEAR scheme requires two parameters, *detection interval* and *group size*. We set the detection interval to half of the stream length, so that DEAR does a single detection over the whole stream. Group size is set to 10. DEAR is quite sensitive to changes in the stream. For example, both (b)(c) and (e)(f) are pairs of similar streams. However DEAR succeeds for one but fails for the other in both cases. As discussed in section 1, the PCC detection scheme tends to mix locality with looping. Here it misclassifies three

non-loop streams as loops. Actually it detects the highly temporally clustered stream (a) as looping.

3.2 Caching Performance

We used trace-driven simulation to study caching performance of AMP. Only a subset of our results are shown here; our full results are presented in [7]. All traces were collected on a 2.4 GHz Pentium 4 Xeon PC server using the tracing functionality of our AMP implementation. One difficulty we encountered was that [4] does not contain a detailed specification of PCC’s partitioned cache manager. Hence, we implemented the PCC pattern detection algorithm and used AMP’s cache management module. We believe this gives a fair evaluation of the detection algorithm because it should be orthogonal to the cache management algorithm. We call this hybrid scheme PCC*.

Figure 2 shows results for using `cscope` to look up 5 symbols in the index (sized 106MB) of Linux kernel source code. Because `cscope` does big looping accesses to the index file, DEAR, PCC* and AMP all perform similarly and much better than LRU and ARC, which see no improvement until the cache is large enough to hold the whole index file.

The trace *scan* (Figure 3) shows a pathological case for PCC*. In this trace, a test program walks the Linux kernel source once and reads each file three times. These “small loops” are classified as “loop” by PCC and MRU is used. AMP classifies these as “temporally clustered” because more recent blocks get accessed. Figure 3 shows that PCC* performs much worse than all others, including LRU.

Figure 4 shows performance while building the Linux kernel. Since the accesses in this trace show a high degree of locality and include many small loops, detection based methods would cannot improve things by applying MRU. In reality, PCC* and DEAR show degradation compared to LRU/ARC. PCC classifies some “small loop” contexts as loops and loses hits. AMP detects these correctly and shows slight improvements over LRU and ARC.

Our final simulation test was of DBT3 [3], an open-source implementation of the commercial TPC-H database benchmark. This a relatively large trace. The whole database is about 4GB, with each query against a large portion of the database. It is run on PostgreSQL 7.4.2. We ran only 16 of the 22 queries because the other 6 take too long to finish (hours to days) due to known issues

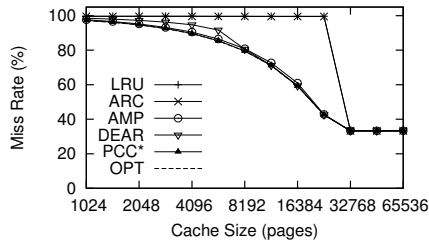


Figure 2: *cscope*

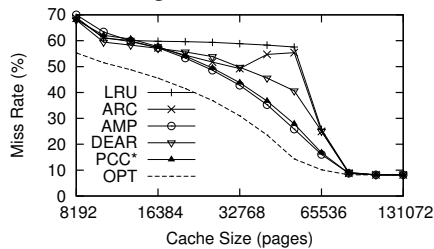


Figure 5: *dbt3*

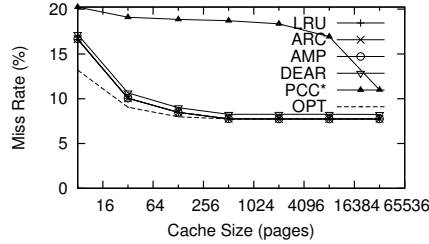


Figure 3: *scan*

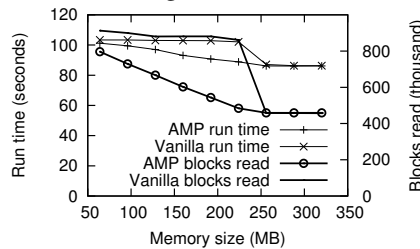


Figure 6: Glimpse on AMP and Linux

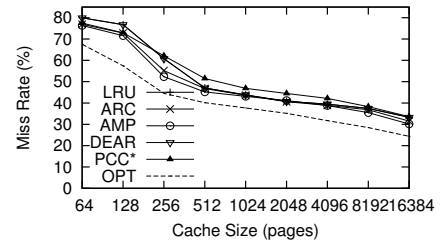


Figure 4: *linuxkernel*

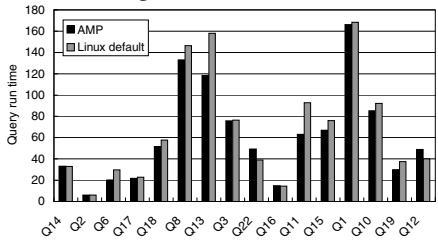


Figure 7: Query exec. times of *dbt3*

with PostgreSQL 7. At over 700M samples, this trace was too large for our simulator. Hence we down-sampled this trace by a factor of 7, reducing its working set to 1/7 of the original. Figure 5 shows detection based methods out-perform ARC/LRU again here. DEAR shows less improvement, probably because the complex mix of accesses from the database process makes DEAR's process-based detection less accurate. AMP and PCC* yield greater improvements and perform similarly. For example, for a cache size of 52016 pages, AMP achieves miss rate of 25.9%, compared with ARC at 55.4% and LRU at 57.6%, a reduction of more than 50%.

3.3 Measurement

Here we compare our AMP implementation with the default Linux page cache by benchmarking real applications. Our test machine was a Pentium 4 Xeon 2.4GHz server with 1GB of memory running Linux 2.6.8.1 with and without our AMP modifications.

Our first test application was *glimpse*, a text-retrieval tool. We use the *glimpseindex* command to index the Linux 2.6.8.1 kernel source files (sized 222MB). The execution times and number of blocks read from disk are shown in Figure 6. AMP shows significant performance improvement over the Linux page cache. Run time decreases by up to 13% and the number of blocks read from disk decreases by up to 43%, both when memory size is 224 MB.

We also ran the DBT3 database workload, in the same configuration as our *dbt3* trace. Figure 7 shows the execution times of queries on AMP and plain Linux. AMP did better in 11 queries and worse in 5 (Q14, Q2, Q8, Q22 and Q12) (reason under investigation). AMP shortens the total execution time by 9.6%, from 1091 seconds to 986 seconds. Disk read traffic decreased by 24.8% from 15.4 to 11.6 GB. Write traffic decreased by 6.5%, from 409 MB to 382 MB, probably due to lower cache contention.

4 Conclusion

We have presented AMP, an adaptive caching system that deduces information about an application's structure and uses it to pick the best cache replacement policy for each program context. Compared to recent and concurrent efforts, AMP is unique in that it uses a low-overhead and robust access pattern detection algorithm, as well as a randomized partition size adaptation algorithm. Simulation confirms the effectiveness and robustness of the pattern detection algorithm. And measurement results on the Linux prototype are promising.

References

- [1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 198–212. ACM Press, 1991.
- [2] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. An implementation study of a detection-based adaptive block replacement. In *Proceedings of the 1999 Annual USENIX Technical Conference*, pages 239–252, 1999.
- [3] OSDL DBT3 database workload. http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/osdl_dbt-3/.
- [4] Chris Gniady, Ali R. Butt, and Y. Charlie Hu. Program counter based pattern classification in buffer caching. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004.
- [5] Jongmin Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping reference. In *Symposium on Operating System Design and Implementation (OSDI'2000)*, 2000.
- [6] Nimrod Megiddo and Dharmendra S. Modha. Outperforming LRU with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.
- [7] Feng Zhou, Rob von Behreh, and Eric Brewer. Program context specific buffer caching with AMP. Technical report, CS Division, University of California, Berkeley, 2005.

Automatic Synthesis of Filters to Discard Buffer Overflow Attacks: A Step Towards Realizing Self-Healing Systems*

Zhenkai Liang, R. Sekar, and Daniel C. DuVarney
Department of Computer Science, Stony Brook University
{zliang, sekar, dand}@cs.sunysb.edu

Abstract

Buffer overflows have become the most common target for network-based attacks. They are also the primary propagation mechanism used by worms. Although many techniques (such as StackGuard) have been developed to protect servers from being compromised by buffer overflow attacks, these techniques cause the server to crash. In the face of automated, repetitive attacks such as those due to worms, these protection mechanisms lead to repeated restarts of the victim application, rendering its service unavailable. In contrast, we present a promising new approach that learns the characteristics of inputs associated with attacks, and filters them out in the future. It can be implemented without changing the server code, or even having access to its source. Since attack-bearing inputs are dropped *before* they corrupt the victim process, there is no need to restart the victim; as a result, recovery from attacks can be very fast. We tested our approach on 8 buffer overflow attacks reported in the past few years on `securityfocus.com` and were available with working exploit code, and found that it generated accurate filters for 7 out of these 8 attacks.

1 Introduction

Self-healing is emerging as an exciting new area within computer security. A key characteristic of approaches in this area is their ability to detect ongoing attacks, identify the underlying vulnerability being exploited, and adapt the system to “heal” the vulnerability. Once healed, the system becomes immune to subsequent attacks that exploit the same vulnerability. An important benefit of self-healing is that it avoids system resources from being spent on reactive defenses, such as system restarts, which can adversely impact system availability.

Although self-healing approaches have been studied in the context of spontaneous faults, they have just begun to receive attention in the context of computer and network security. We present a new approach that represents an important first step towards realizing *practical* defenses that employ self-healing. Our approach focuses on buffer

overflows, which have become the most common target for network-based attacks. Among the COTS-related security advisories released by CERT Coordination Center in 2003 to 2004, 41 of the 51 were related to buffer overflows. Moreover, they are the primary mechanism used by worms in order to propagate.

The state-of-art in defenses against buffer overflows includes various *guarding* techniques [3, 4] for preventing execution from data segments, and *randomization* techniques [1, 2]. Although these techniques can detect attacks before vital system resources (such as files) are compromised, they cannot protect the victim process itself, whose integrity is compromised prior to the time of detection. For this reason, the safest approach for recovery is to terminate the victim process. With repetitive attacks, such as those due to worms, or other forms of automated attacks, these approaches will cause repeated server restarts, effectively rendering a service unavailable during periods of attack. In contrast, our self-healing approach can filter out attacks *before* process integrity is compromised, thereby enabling the service to continue without any interruption. Moreover, our approach doesn’t require any user-supplied knowledge about the server, or access to its source code.

2 Overview of Approach

Our approach, called ARBOR (Adaptive Response to Buffer Overflows), is designed to protect network server processes. It is based on the observation that attacks arrive via inputs to these processes. Figure 1 illustrates the architecture of ARBOR, which forms a protective layer between a process and the external environment by adaptively filtering out attack inputs. The adaptation is based on a feedback loop: inputs which don’t trigger an intrusion report from the detector are allowed to pass through the filter unmodified, while inputs that trigger an intrusion report activate the feedback loop, causing the filter to be modified to block future attacks. The principal components of ARBOR are described below.

The *input filter* inspects all input entering the system and filters out (i.e., drops) those inputs that match existing filter rules. The filtering rules are generated (auto-

*This research is supported in part by an ONR grant N000140110967 and an NSF grant CCR-0208877.

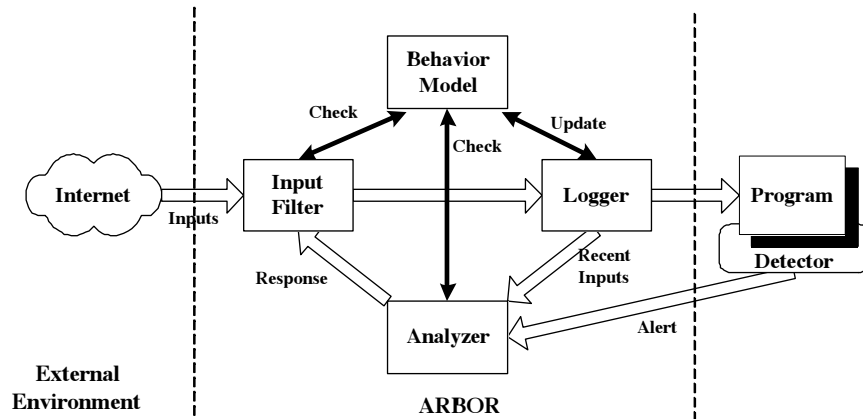


Figure 1: Architecture of our approach, which is a protective layer between the protected program and the external environment.

matically) by the *analyzer* component, and are intended to capture characteristics that distinguish attack-bearing inputs from benign ones.

The *behavior model* is a central component that is consulted by all components of ARBOR. It is constructed from events, such as system calls and other relevant library calls, intercepted by the logger. The model takes the form of a finite-state automaton that is similar to a control-flow graph of a program, except that (a) it records only those events reported by the logger, and not the internal program actions, such as assignments and jumps, and (b) it captures only those program paths that were actually traversed. The behavior model is based on our earlier work [7]. The behavior model provides contextual clues that form the basis of the filter generation logic in the analyzer component, as well as provides the tests made by the input filter.

The *logger*, like the rest of the components, is implemented using library call interposition. (Instead of system call interposition, we use library interposition because of its low performance overhead. Library interposition also provides adequate security for our purposes, since we are interested in program behaviors before it is compromised.) It records a log of input events that is used subsequently by the analyzer. In order to reduce space as well as time overheads due to logging, only a subset of data is sampled and logged.

The *detector* is external to ARBOR. Our current implementation uses address obfuscation [2] to build the detector. With the defense of address obfuscation, each buffer overflow attempt will be turned into a server crash. We intercept this crash event to trigger the feedback loop in ARBOR.

The *analyzer* is responsible for synthesizing filters that distinguish attack-bearing inputs from benign ones. The analyzer resides in a separate process from the protected

process, collecting recent program behavior events from the logger. If the protected process is attacked, the analyzer will be notified by the detector. Upon receiving a notification, the analyzer examines recent inputs to the protected process to identify attack-bearing inputs. The process of analysis is described in greater detail in the following section.

2.1 Automatic Identification of Attack-bearing Inputs

Given the nature of buffer overflow attacks, a basic characteristic for filter generation come to mind — length of input data, as buffer overflow attacks are usually associated with receiving inputs that are longer than what is expected by the program. But length of input alone is not sufficiently powerful to identify inputs of typical buffer overflow attacks, because an attacker only needs to provide an input longer than the normal input for that buffer, not necessarily longer than all other inputs to the program. Therefore, the attack-bearing input may not be distinguishable among all the inputs. However, if we compare the inputs serving the same purpose, the length difference between attack-bearing inputs and benign inputs will likely become more obvious. In our approach, we use the program’s execution context of input operations to provide us “hints” about the purpose of an input. The problem, then, is how to extract context information from a program’s execution.

One of the key insights in this paper is that we can infer relevant context information by observing the actions of the protected process, e.g., the execution path taken by the program, the contents of runtime stack at the point of input operation, parameters to input operation. We treat a process as a state machine, which makes transitions from one state to another. The transitions are made on library calls. In our approach, we use the location in the program

from which library calls are made to represent states of the state machine. The state machine provides a context of the process's current operation. In addition, we may rely on context information observed near the input operation in question. This leads to two major types of contexts: *current context* and *historical context*.

Current context is the program state at the point of input, which helps to distinguish a specific input operation from others made by the protected program. In our implementation, current context is defined by the location in the program from which the input operation was invoked, and a sequence of return addresses on the program's stack. The return addresses provides information about how the current operation is invoked. It is particularly helpful when the program uses a centralized input handler function that is called from multiple places in the code, and this function in turn invokes the actual input operation. In this case, the calling location for the input function may always remain the same, but the sequences of return address on the stack are different.

With the current context, the analyzer synthesizes a filter rule matching the attack as follows. For each suspicious input, the analyzer first identifies its current context C , and then retrieves the input statistics under context C , which is maintained by the logger during the program's normal execution. If in this context, the size a of the suspicious input is significantly larger than the maximum size b_{max} that has ever been seen during normal execution, we report this input as an attack-bearing input. The synthesized filtering rule is simply one that flags an attack if the input size is larger than the geometric mean of a and b_{max} .

If the context of all input operations made by a program are identical, then the above approach may fail to distinguish between attack-bearing and benign inputs. In this case, we extend our approach to use *historical context*, which takes into account the program paths that were taken prior to the input operation.

2.2 Light-weight Recovery After Discarding Attack-bearing Inputs

After discarding input, it is necessary for the server process to take recovery actions, so that it can get ready to serve future requests. However, it is difficult to *automatically* discover the set of actions to be taken. To address this problem, we observe that networked servers expect and handle transient network errors, which can cause input operations to fail. We leverage this error recovery code to perform the necessary clean up actions. Specifically, whenever ARBOR drops attack-bearing input, it changes the return code of the input operation to an error code associated with a network error. This error code causes the server to invoke its recovery code, including freeing of resources allocated to process the client re-

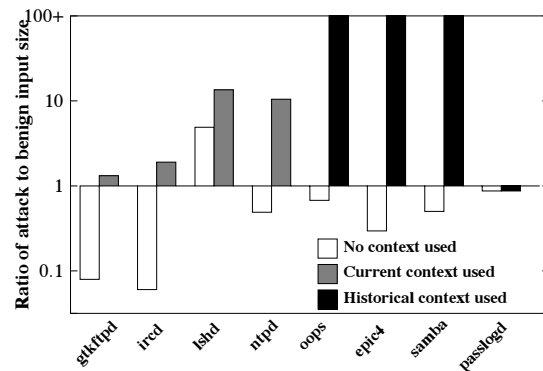


Figure 2: Effectiveness of ARBOR in synthesizing size-based filters against buffer overflow attacks. A value less than one indicates that ARBOR is not able to distinguish attack-bearing inputs from benign input.

quest and so on. We have found this approach to work successfully in all of our experiments.

3 Preliminary Results

We used ARBOR to defend several programs against remote buffer overflow attacks. Our focus was on “real” buffer overflow attacks, so we examined buffer overflow attacks reported on securityfocus.com during 2001–2003. We found eight attacks with working exploit code.

Figure 2 shows the results obtained with these programs, which are ratios of attack-bearing input size and maximum benign input size. A ratio less than one indicates that ARBOR cannot distinguish attack-bearing input from benign input under the program context used. The results are divided into three groups according to the context involved in the synthesized filters. In the first group (gkftpd, ired, lshd, ntpd), current context (specifically, the program location from where the input operation was called) was enough to generate an effective filter. The programs in the second group (oops, epic4, samba) received several types of inputs from a single location, so current context was not sufficient to synthesize filters. However, by using historical context, ARBOR was able to create an accurate filter that discarded subsequent attacks. The third group (passlogd) consists of a single program, for which ARBOR cannot successfully generate a filter. The buffer overflow in the third group is caused by part of the input, which cannot be identified by the overall length of the input.

Importance of context information As we can see from the figure, when context information is not involved, only 13% (1 out of 8) of attacks can be identified, in which the attack-bearing input is so huge that it is larger than all other inputs. After current context is involved, 50% of the attacks can be identified and fil-

tered out. If both current context and historical context are used, ARBOR can generate effective filters for 88% of the attacks.

As a final point, we note that we restricted ourselves to length-based filters as a way to stress our approach for inferring and using program context in filtering rules. By using other criteria, such as input character distributions, in conjunction with length-based and context-based filtering rules, the approach can be made even more effective.

4 Related Work

Shield [9] is also aimed at filtering out network-based attacks on servers. Whereas our approach synthesizes filters automatically for a subclass of attacks (buffer overflows), Shield is based on manually developed filtering rules to address a broader range of attacks.

Automatic patch generation [8] attempts to deal with rapidly propagating worms by automatically generating a patch to fix the vulnerability being exploited by the worm. The primary differences with our approach are that [8] uses a more complex generate-and-test approach to diagnose the vulnerability exploited in an attack, requires source code of the protected server, plus an isolated, sandboxed duplicate of the protected server to test the correctness of the patch.

The *HACQIT* project [5] takes an alternative approach that combines software diversity with content-filters deployed at the network level. Attacks are suspected when two implementations of the same software yield different results on the same input. A rule-based algorithm is used to learn characteristics of inputs suspected of containing attacks, and generate a filter to discard such requests in the future at the firewall. This algorithm has been shown to work against Code Red worm. However, it is not clear how the algorithm can be generalized to deal with all types of buffer overflow attacks.

Failure-oblivious computing [6] is a source code transformation approach that can also recover quickly from attacks. It detects out-of-bounds write accesses, and directs them to a different (free) memory area. Subsequent out-of-bounds reads to the same area return the data that was previously written. Other out-of-bounds reads return carefully chosen values, e.g., all zeroes. The main strength of this approach is that it reliably detects the root cause of the problem. Its weaknesses are the high overheads required for memory error detection, often exceeding 100%; and the possibility that processing the attack input may cause the program to fail and/or crash, although their experiments indicate that is atypical.

5 Discussion

Our preliminary results demonstrate the feasibility of developing self-healing approaches to protect servers from

buffer overflow attacks. Unlike previous approaches that were focused mainly on preventing a system compromise, our approach is able to provide complete immunity, in the sense that attacks don't even have a performance impact on the victim server.

While the approach is very effective on existing attacks, it does have some weaknesses that may be exploited in attacks specifically designed to fool it. For instance, attacks may be delivered through a sequence of small packets, causing each input operation to return a small amount of data. We are developing techniques to deal with this problem by assembling together the data returned by successive input operations, and developing filters based on this assembled data rather than the data returned by individual input operations.

A second problem concerns attacks where the buffer overflow is triggered by a field in the request, and not the entire request. Therefore, the length of input is not a characteristic of the attack. In this case, we need to identify the vulnerable field in the input, and use the identification information to increase the accuracy of program contexts. We are currently investigating techniques to handle such attacks.

References

- [1] The pax team. <http://pax.grsecurity.net>.
- [2] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of 12th USENIX Security Symposium*, August 2003.
- [3] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of 7th USENIX Security Conference*, January 1998.
- [4] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL <http://www.trl.ibm.com/projects/security/ssp/main.html>, June 2000.
- [5] J. C. Reynolds, J. Just, L. Clough, and R. Maglich. On-line intrusion detection and attack prevention using diversity, generate-and-test, and generalization. In *Proceedings of the 36th Hawaii International Conference on System Sciences*, 2003.
- [6] M. Rinard, C. Cadar, D. Roy, and D. Dumitran. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, December 2004.
- [7] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2001.
- [8] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. Technical Report CUCS-029-03, Columbia University Department of Computer Science, 2003.
- [9] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, August 2004.

Facilitating the Development of Soft Devices

Andrew Warfield, Steven Hand, Keir Fraser and Tim Deegan

University of Cambridge Computer Laboratory, J J Thomson Avenue, Cambridge, UK
{firstname.lastname}@cl.cam.ac.uk

1 Introduction

Device-level interfaces in operating systems present a very useful cut-point for researchers to experiment with new ideas. By virtualizing these interfaces, developers can create *soft devices*, which are used in the same way as normal hardware devices, but provide extra functionality in software. Recent years have shown this approach to be of considerable interest: a few examples of block device extension include the addition of intrusion detection systems to disk interfaces [1], the development of “semantically smart” disks [2], and that of time-travel block devices [3]. Other devices, such as network interfaces, have similarly been extended.

Working at the device interface allows an examination of the functional separation between hardware and software: researchers can simulate new features as if they were properties of the device itself. As simple examples, block or network device interfaces might be extended to compress or encrypt data before it is written to disk or transmitted. Alternatively, it may be desirable to prototype entirely new devices in software, bound to existing interfaces, for instance a content-addressable disk.

Unfortunately, researchers face a challenge in extending devices in this manner. Implementors must typically modify an existing operating system to add the new functionality, often by creating OS-specific pseudo-devices. This requirement means learning the OS source and writing scaffolding code to intercept events. Moreover, where new functionality must be developed in-kernel it is difficult to debug and crashes are not contained. Finally, these low-level developments are difficult to share and maintain across systems, as they will be specific to the OS, or even specific version thereof, that it has been developed within.

This paper presents a solution to the problems associated with developing soft devices by extending the existing device interface in Xen [4]. Xen is a virtual machine monitor (VMM) for the IA32 architecture that *paravirtualizes* hard-

ware: Rather than attempting to present a fully virtualized hardware interface to each OS in a Xen environment, guest OSes are modified to use a simple, narrow and idealized view of hardware. Soft devices take advantage of these narrow interface to capture and transform block requests, network packets, and USB messages.

As an initial example of this approach, we have implemented a *block tap*, which is an interface to facilitate the development of soft devices for block device access. The block tap allows soft devices to be constructed as user-space applications in an entirely isolated virtual machine. This strong isolation from the remainder of the system allows a single soft device to work with any OS and hardware available on Xen, and allows developers to work with high-level languages and debuggers. While our approach aims to facilitate development it still provides a high level of performance, sustaining 50MB/s read throughput for disk requests in our experiments.

2 Device Access in Xen

The existing approach to device access in Xen makes use of *split device drivers*, and has been described in detail in [5]. Figure 1 illustrates a split driver: A device driver VM is granted specific access to the physical hardware that it will manage. This VM runs an existing, unmodified (e.g. Linux) device driver to access the device. In addition it runs a *back-end* driver, which provides a simple narrow interface to the device. Operating systems wishing to access the device will use a *front-end* driver, and interact with the back-end over a device channel, which is a shared-memory communications primitive.

This approach aims to improve system stability while still supporting existing device drivers by isolating drivers in a single VM, away from both other OSes and the VMM. Perhaps more importantly though, the split driver interface simplifies device access for operating systems above Xen, as an OS need only implement a single front-end driver to

Split Device Drivers in Xen

Physical driver runs in an isolated VM, connected over a shared memory device channel to a guest VM accessing the device.

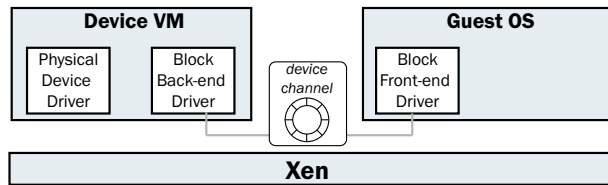


Figure 1: Split device drivers in Xen

support an entire class of devices (e.g. block storage).

Xen's current split block device illustrates exactly how narrow this interface is. The device channel shown in Figure 1 is a single page of memory shared between the two VMs. A bi-directional ring buffer is used to pass messages. Pages of data are attached and mapped separately. The interface has only three commands: READ, WRITE, and PROBE. READ and WRITE provide block-level access to data, while PROBE returns a list of accessible block devices.

3 Implementing a soft device Interface

This section describes how we have extended Xen's existing split device interface to support the development of soft devices. We describe an implementation of a block tap that allows the construction block soft devices. In designing and implementing the block tap, we have attempted to meet three general requirements:

1. **Make device implementation easy.** Our principal goal is to facilitate the development and exploration of new functionality for device interfaces. We have chosen to give developers the option of receiving device requests through a user-level interface in the soft device domain, allowing development in a safe environment with a variety of languages and tools.
2. **Do not modify the existing guest OS or device VM.** While Xen itself achieves performance by modifying the guest operating system, we desire that the soft device interface leave the attached front-end and back-end VMs unmodified. This approach allows the development of soft devices between any hardware and OS combination supported by Xen without necessitating the modification (or even understanding) of that code. Additionally, the soft device interface may be used to trace, debug, or modify an existing split device connection. As such, all soft device implementation exists within an isolated VM using only the shared-memory device channels as an interface.
3. **Maintain satisfactory performance.** We hope to use soft devices in practical situations, and not just as a

Block Tap Device Structure

The application accesses the block device interface to generate and receive block messages. The message switch has a variety of forwarding modes.

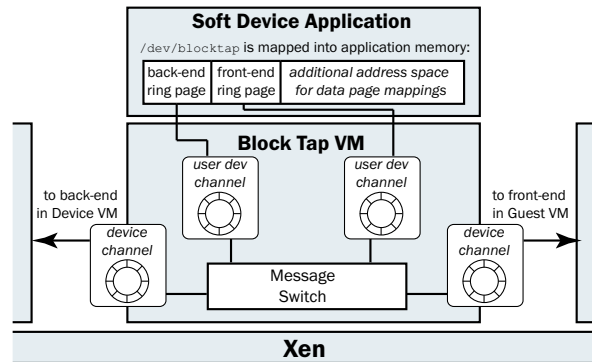


Figure 2: block tap structure.

prototyping tool. By taking advantage of the performance allowed through request batching, we hope to maintain a high level of throughput for soft devices.

The remainder of this section describes our block tap implementation addressing these requirements. The block tap is currently about 1300 lines of commented C code, and runs in a Linux-based VM.

3.1 A Switch for Block Requests

The potential ways in which a soft device interface might be used are varied. Developers may desire to simply trace request traffic in order to monitor usage patterns, they may wish to modify in-flight requests, or they may desire to construct a terminating device, which does not forward requests at all.

In order to accommodate these varied modes of operation, we have implemented the soft device interface as a request switch. The driver is plumbed into the device channel between the front-end and back-end domains. In this position, all block requests pass through it.

The new driver acts as a switch, shown in Figure 2, forwarding messages across four rings. Two of these rings are inter-VM shared memory rings as described above. They connect to the front-end and back-end drivers that the soft device interface has been placed between. Two additional rings extend from the block tap up to application space in the same VM. These rings are accessed through a character device that can be mapped by applications.

An `ioctl()` to this character device is used to set the switching mode used for block messages. Three common modes are described here, and shown in Figure 3.

`MODE_PASSTHROUGH` is the lowest-overhead switching configuration. In this mode, messages are passed straight through the driver on to the opposite ring, and completely

Examples of Forwarding Modes in the Block Tap

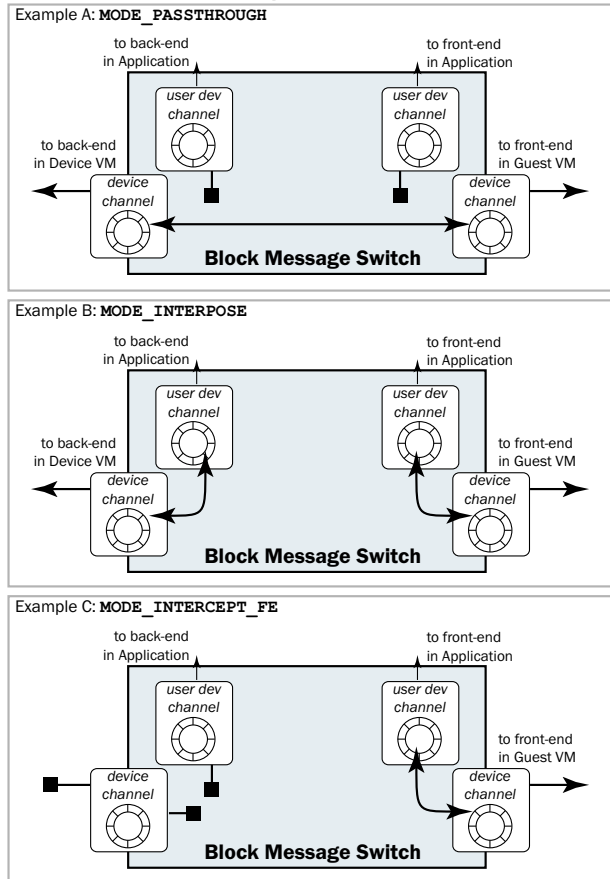


Figure 3: Examples of forwarding modes.

bypass the user rings. Passthrough can be used to implement kernel-level monitoring of block requests, or to implement soft devices in-kernel for improved performance.

MODE_INTERPOSE routes all requests and replies across the user rings. An application must attach to the block tap interface and pass messages across the two rings, allowing complete monitoring and modification of the request stream at the application level. This mode can be used to modify in-flight requests, for instance to build a compressed or encrypted block store.

MODE_INTERCEPT_FE uses only the front-end rings on the driver, disabling the back-end altogether. This mode allows the construction of full, application-level soft devices, using existing OS interfaces (such as memory, or mounted file systems) as a backing store. This mode can be used to easily prototype new functionality, or to forward block requests to a block device back-end on another physical host (after an OS migration, for instance¹).

¹OS migration is feature that we have recently added to Xen, allowing a running OS to move from one physical host to another while executing. One problem which managing migration is that local disks will be left behind.

3.2 The Application Interface

As shown in Figure 2, the user rings are exported to a character device, which is mapped by a library allowing access to the message rings and in-flight requests. Our current implementation allows chains of plugins to be attached to handle block requests. We presently have plugins to provide both copy-on-write and encrypted disks and to allow direct access to image files and remote GNBD disks.

4 Evaluation

Figure 4 shows an analysis of the impact of soft devices on block request performance with respect to both throughput and latency. Tests were performed on a Compaq Proliant DL360, which is a dual Pentium III 733MHz machine with 72.8GB Ultra3 SCSI disks.

Throughput measurements aimed to test the maximum achievable read and write speeds to the local disk. The left graph in Figure 4 shows read and write throughput moving four gigabytes of sequential data to and from disk. The three bars in the graph compare the throughput without using the block tap, using the block tap in MODE_PASSTHROUGH, and finally in MODE_INTERPOSE. As shown, our soft device interface results in a minimal degradation of throughput. We are capable of achieving 50MB/s read throughput, identical to that achieved by Xen's existing split drivers. On writes, we see about a 15% overhead; we are still investigating the source of this loss of performance.

Latency measures the per-request overhead of synchronous requests to disk. Given that disk requests are heavily batched in general, this is a less meaningful measurement for normal workloads. However, it does represent a worst-case overhead and also gives a clearer illustration of the costs that our implementation imposes. The right graph in Figure 4 shows mean request times across 100,000 4-byte synchronous writes. We see a small overhead in passing requests through the kernel of the virtual device domain in MODE_PASSTHROUGH, reflecting the cost of an additional VM context switch and request/response copy² in each direction. MODE_INTERPOSE is considerably more expensive as it adds two additional context switches and two message copies, in order to pass messages through a user-space application. There are additional costs in mapping attached data pages to user space. However, even this overhead has insignificant impact given the length of average disk seek times. We intend to explore the more demanding performance requirements of network devices in the coming months.

²Note that only the request and response structs (respectively 60 and 7 bytes) are copied on the shared memory rings. Pages of data are referenced and mapped separately.

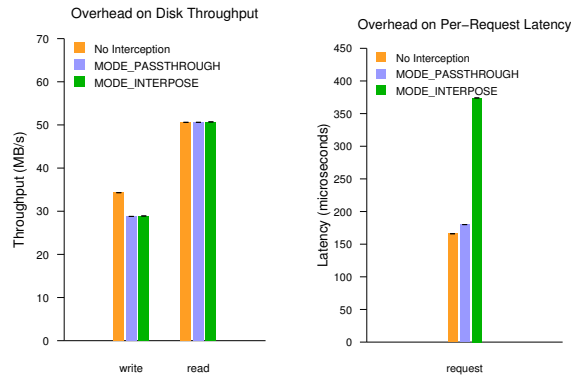


Figure 4: Overhead of virtual block devices.

5 Related and Future Work

The use of virtual machines to provide device extensibility has been explored previously in μ Denali [6]. While our intentions of facilitating development of soft devices are identical, our system is very different. μ Denali is a VMM hosting BSD VMs, which access devices over micro-kernel-style IPC through Mach's port abstractions. Xen does not use synchronous micro-kernel-style IPC; instead it detaches message transfer (copying/mapping pages between VMs) from notification (virtual interrupts), to achieve high throughput through message batching. We feel that these differences are interesting for two reasons: first, the design presented here represents a considerably different approach to that presented in [6]. Second, as Xen is a publicly available VMM and supports the production use of several popular OSes, we hope that the availability of this work is of general interest to other researchers.

The block tap itself is similar in some ways to the FreeBSD GEOM and Linux EVMS projects, both of which provide a great deal of extensibility to their respective systems' block device interfaces. The block tap aims to provide similar extensibility, but in a virtualized environment, thus catering to a range of operating systems. The use of virtualization also allows strong isolation for soft devices, enhancing stability.

We view the soft device interface as an enabling tool for future work. As mentioned earlier, other projects have explored the use of interposition on device interfaces for such goals as intrusion detection [1], semantically smart disks [2], and fault diagnosis [3]. Two projects that we are particularly interested in exploring over these interfaces in the coming months are described here.

Parallax - a cluster storage system for virtual hosts. Managed virtual machines present different file system requirements than are currently available through distributed or cluster file systems, or network-attached storage. We are

currently building a cluster-based storage system to provide high-availability storage for virtual OSes that may migrate between physical hosts, return to arbitrary historical snapshots, and which likely exhibit a large amount of commonality in terms of their system images.

Data-stream watchpoints for pervasive debugging. Interposing on block and network requests will allow existing OS debugger work above Xen to be extended to enable break and watch points on specific content. We intend to allow debugging to be triggered if a specific file is modified, or certain network traffic is sent or received.

6 Conclusion

This paper has briefly presented the implementation of a virtual device interface for OSes on the Xen VMM. Our implementation enables the construction of new functionality to trace, interpose on, or extend existing block device interfaces. We look forward to extending this approach to support network and USB devices, and to building systems above it.

References

- [1] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the USENIX Security Symposium.*, August 2003.
- [2] M. Sivathanu, V. Prabhakaran, F. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, March 2003.
- [3] A. Whitaker, R. S. Cox, and S. D. Gribble. System administration as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04)*, December 2004.
- [4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP19)*, pages 164–177. ACM Press, 2003.
- [5] K. Fraser, S. Hand, I. Pratt, A. Warfield, R. Neugebauer, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the First Workshop on Operating System and Architectural Support for the on-demand IT Infrastructure (OASIS-2004)*, October 2004.
- [6] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

Implementing Transparent Shared Memory on Clusters Using Virtual Machines

Matthew Chapman and Gernot Heiser

The University of New South Wales, Sydney, Australia

National ICT Australia, Sydney, Australia

matthewc@cse.unsw.edu.au

Abstract

Shared memory systems, such as SMP and ccNUMA topologies, simplify programming and administration. On the other hand, clusters of individual workstations are commonly used due to cost and scalability considerations.

We have developed a virtual-machine-based solution, dubbed vNUMA, that seeks to provide a NUMA-like environment on a commodity cluster, with a single operating system instance and transparent shared memory. In this paper we present the design of vNUMA and some preliminary evaluation.

1 Introduction

Many workloads require more processing power than feasible with a single processor. Shared-memory multiprocessors, such as SMP and NUMA systems, tend to be easier to use, administer and program than networks of workstations. Such shared-memory systems often use a *single system image*, with a single operating system instance presenting a single interface and namespace. On the other hand, clusters of individual workstations tend to be a more cost-effective solution, and are easier to scale and reconfigure.

Various techniques have been proposed to provide the simplicity of shared-memory programming on networks of workstations. Most depend on simulating shared memory in software by using virtual memory paging, known as *distributed shared memory* (DSM) [1]. At the middleware layer there are DSM libraries available, such as Treadmarks [2]. These libraries require software to be explicitly written to utilise them, and they do not provide other facets of a single system image such as transparent thread migration. Some projects have attempted to retrofit distribution into existing operating systems, such as the MOSIX clustering software for Linux [3]. However, Linux was not designed with such distribution in mind, and while MOSIX can provide thread migration, many system calls still need to be routed back to the original node. Other projects have attempted to build distributed operating systems from the ground up, such as Amoeba [4] and Mungi [5]. In order to gain wide ac-

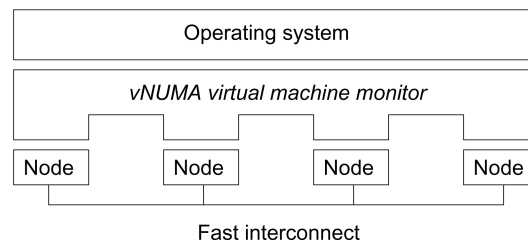


Figure 1: Cluster with vNUMA

ceptance, these operating systems need to provide compatibility with a large body of existing UNIX applications, which is no easy task.

In this paper, we present an alternative approach utilising virtualisation techniques. Virtualisation can be useful for hiding hardware complexities from an operating system. A privileged *virtual machine monitor* interposes between the operating system and the hardware, presenting virtual hardware that may be different from the real hardware. For example, Disco [6] simulates multiple virtual SMP systems on a NUMA system.

vNUMA uses virtualisation to do essentially the opposite — simulating a single virtual NUMA machine on multiple workstations, using DSM techniques to provide shared memory. Unlike previous work, this can achieve a true single system image using a legacy operating system, without significant modifications to that operating system.

We focus on Linux as the guest operating system, since it supports NUMA hardware and the source code is available. This means that there are already some optimisations to improve locality, and we can make further improvements if necessary.

We chose to target the Itanium architecture [7] for our virtual machine. Numerous IA-32 virtual machine monitors already exist, and a number of the techniques are encumbered by patents. Itanium is being positioned by Intel as the next “industry standard architecture”, particularly for high-end systems. An Itanium virtual machine monitor presents some research opportunities in itself, independent of the distribution aspects.

2 Implementation overview

2.1 Startup

In order to achieve the best possible performance, vNUMA is a *type I* VMM; that is, it executes at the lowest system software level, without the support of an operating system. It is started directly from the bootloader, initialises devices and installs its own set of exception handlers.

One of the nodes in the cluster is selected as the bootstrap node, by providing it with a guest kernel as part of the bootloader configuration. When the bootstrap node starts, it relocates the kernel into the virtual machine's address space, and branches to the start address; all further interaction with the virtual machine is via exceptions. The other nodes wait until the startup node provides a start address, then they too branch to the guest kernel; its code and data is fetched lazily via the DSM.

2.2 Privileged instruction emulation

In order to ensure that the virtual machine cannot be bypassed, the guest operating system is demoted to an unprivileged privilege level. Privileged instructions then fault to the virtual machine monitor. The VMM must read the current instruction from memory, decode it, and emulate its effects with respect to the virtual machine. For example, if the instruction at the instruction pointer is `mov r16=psr`, the simulated PSR register is copied into the userspace `r16` register. The instruction pointer is then incremented.

The Itanium architecture is not perfectly virtualisable in this way and has a number of sensitive instructions, which do not fault but require VMM intervention [8, 9]. These must be substituted with faulting instructions. Currently this is done statically at compilation time, although it would be possible to do at runtime if necessary, since the replacement instructions are chosen so that they fit into the original instruction slots. The `cover` instruction is simply replaced by `break`. `thash` and `ttag` are replaced by moves from and to *model-specific registers* (since model-specific registers should not normally be used by the operating system, and these instructions conveniently take two register operands).

2.3 Distributed Shared Memory

The virtual machine itself has a simulated physical address space, referred to here as the machine address space. This is the level at which DSM operates in vNUMA. Each machine page has associated protection bits and other metadata maintained by the DSM system. When the guest OS establishes a virtual mapping, the effective protection bits on the virtual mapping are calculated as the logical AND of the requested protection bits and the DSM protection bits. vNUMA keeps track

of the virtual mappings for each machine page, such that when the protection bits are updated by the DSM system, any virtual mappings are updated as well.

The initial DSM algorithm is a simple sequentially consistent, multiple-reader/single-writer algorithm, based on that used in IVY [10] and other systems. The machine pages of the virtual machine are divided between the nodes, such that each node manages a subset of the pages. When a node faults on a page, the manager node is contacted in the first instance. The manager node then forwards to the owner (if it is not itself the owner), and the owner returns the data directly to the requesting node. The copyset is sent along with the data, and if necessary the receiving node performs any invalidations. Version numbers are used to avoid re-sending unchanged page data.

3 Evaluation

Our test environment consists of two single-processor 733Mhz Itanium 1 workstations with D-Link DGE-500T Gigabit Ethernet cards, connected back-to-back with a crossover cable to form a two-processor cluster. We also used a similar dual-processor (SMP) Itanium workstation for comparison. Obviously it is intended that the system will scale beyond two nodes, however the software was not yet stable enough for benchmarking on a larger cluster.

As the guest kernel, we used a Linux 2.6.7 kernel compiled for the HP simulator platform. The only modifications to the kernel are a tiny change to enable SMP (since the HP simulator is usually uniprocessor), and the static instruction replacement described in section 2.2.

The SPLASH-2 benchmarks [11] are a well-known set of benchmarks for shared memory machines. We used an existing implementation designed to work with the standard *pthread*s threading library. Here we present results from three of the SPLASH-2 applications: **Ocean**, **Water-Nsquared** and **Barnes**. In each case we measured the performance on four different topologies: a single-processor workstation, a single-processor workstation with vNUMA (to measure virtual machine overhead), two single-processor workstations with vNUMA, and a dual-processor SMP workstation. We used the processor cycle counter to obtain timings, since we did not want to place trust in the accuracy of `gettimeofday` on the virtual machine.

3.1 Ocean

Ocean simulates large-scale ocean movements by solving partial differential equations. The grid representing the ocean is partitioned between processors. At each iteration the computation performed on each element of the grid requires the values of its four neighbours, causing communication at partition boundaries.

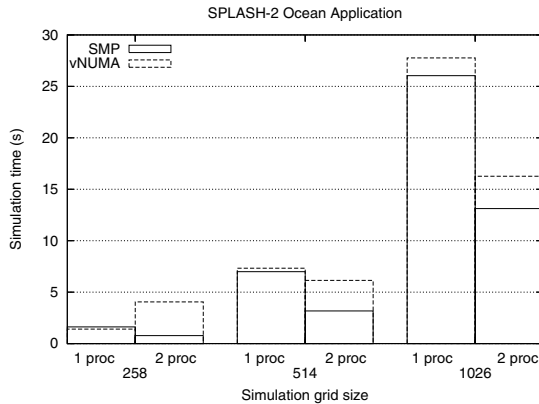


Figure 2: Results of **Ocean** application

The results are shown in Figure 2. First consider the single processor results, which demonstrate virtual machine performance independent of the DSM. At the smallest grid size, 258x258, the virtual machine performance is very good, in fact the benchmark runs marginally faster than without the virtual machine. This is due to the fact that parts of the memory management are done by the virtual machine monitor without involving the guest operating system, and the mechanisms implemented in vNUMA (such as the long format VHPT) are advantageous for some workloads compared to those implemented in Linux [12]. As the grid size and hence working set size increases, the number of TLB misses and page faults that must involve the guest kernel increases. Since these are significantly more expensive on the virtual machine, they ultimately outweigh any memory management improvements. At the largest grid size the virtual machine imposes a 7% overhead.

On the other hand, the distribution efficiency increases with problem size. If the granularity was word-based, communication should increase linearly with one side of the grid. However, because of sparse access patterns compared to the granularity of the DSM, we simply see greater utilisation of the pages being transferred, and the overhead remains roughly constant, meaning that the relative overhead is less. For the 258x258 grid, the vNUMA overhead is significant compared to the actual amount of work being done, and it is clearly not worthwhile. By 514x514, we have passed the “break-even” point and the two-node vNUMA performs better than a single processor. For the largest problem size, the benchmark is largely computation-bound and vNUMA works well. Relative to a single-processor workstation, the vNUMA speedup is 1.60, compared to 1.98 for SMP.

3.2 Water-Nsqared

Water-Nsqared is an example of an application that performs well in a DSM environment [13], and indeed it

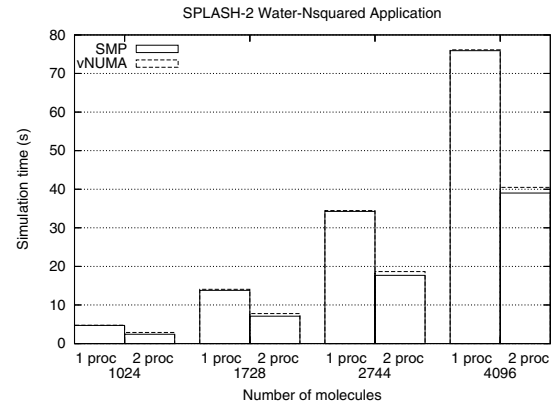


Figure 3: Results of **Water-Nsqared** application

also performs well on vNUMA. **Water-Nsqared** evaluates forces and potentials that occur over time in a system of water molecules. Each processor needs all of the data, but only does a subset of the calculations and stores the results locally. At the end of each timestep, processors accumulate their results into the shared copy. Thus there are alternating read-sharing and update phases.

The results are shown in Figure 3. Here the virtual machine overhead is minimal, since the working set sizes are much smaller than for **Ocean** (around 4MB at the largest problem size, compared to over 220MB). The distribution overhead scales with the number of molecules (and hence the size of the shared data), as might be expected, but again it is small. For the largest problem size, the vNUMA speedup is 1.87, compared to 1.95 for SMP.

3.3 Barnes

On the other hand, **Barnes** is an example of an application that is known not to perform as well in DSM environments [13]. **Barnes** simulates the gravitational interaction of a system of bodies in three dimensions using the Barnes-Hut hierarchical N-body method. The data is represented as an octree with leaves containing information on each body and internal nodes representing space cells. Thus there are two stages in each timestep — calculating forces and updating particle positions in the octree.

The results are shown in Figure 4. The force calculation phase distributes fairly well, certainly for larger problem sizes. However the tree update does not — in this phase the pattern of both reads and writes is fine-grained and unpredictable, which results in significant false sharing. False sharing is particularly problematic because vNUMA currently uses a sequentially consistent, multiple-reader/single-writer DSM, which means pages cannot simultaneously be writable on multiple nodes. Thus, overall, the benchmark does not perform

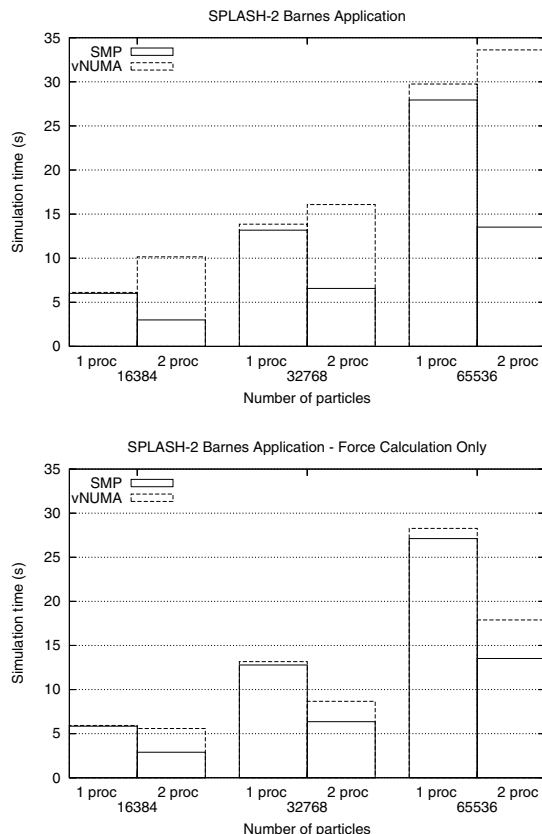


Figure 4: Results of **Barnes** application

well on vNUMA.

4 Conclusions

These results show that, at least for scientific applications such as those in the SPLASH-2 suite, vNUMA performance can be surprisingly good and is dominated by application DSM costs rather than virtualisation or kernel paging overheads. Applications that behave well on conventional DSM systems, such as **Water-Nsquared**, perform best on vNUMA. These are typically applications which are computation-intensive and share pages mostly for reading rather than writing.

However vNUMA has significant advantages over middleware DSM systems, providing a true single system image and a simple migration path for SMP applications. Since it utilises networks of commodity workstations, it is more cost-effective and reconfigurable than specialised ccNUMA hardware. We believe that, at least for some classes of applications, vNUMA could provide a useful alternative to these systems. There are still improvements to be made, and we need to perform benchmarks on larger clusters to prove scalability.

5 Acknowledgements

This work was supported by a Linkage Grant from the Australian Research Council (ARC) and a grant from HP Company via the Gelato.org project, as well as hardware from HP and Intel. National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology and the Arts and the ARC through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

References

- [1] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. Phd thesis, Yale Univ., Dept. of Computer Science, 1986. RR-492.
- [2] P. Keleher, S. Dwarkadas, A. L. Cox, W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, p 115–131, 1994.
- [3] A. Barak, O. La'adan, A. Shiloh. Scalable cluster computing with MOSIX for Linux. In *Proceedings of the 5th Annual Linux Expo*, p 95–100, 1999.
- [4] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, 1990.
- [5] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, J. Liedtke. The Mungi single-address-space operating system. *Softw.: Pract. & Exp.*, 28(9):901–928, Jul 1998.
- [6] E. Bugnion, S. Devine, M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proc. 16th SOSP*, p 27–37, 1997.
- [7] Intel Corp. *Itanium Architecture Software Developer's Manual*, Oct 2002. <http://developer.intel.com/design/itanium/family/>.
- [8] C. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, G. Heiser. Itanium —a system implementor's tale. In *Proc. 2005 USENIX Techn. Conf.*, Anaheim, CA, USA, Apr 2005.
- [9] D. J. Magenheimer, T. W. Christian. vBlades: Optimised paravirtualisation for the Itanium processor family. In *Proc. 3rd Virtual Machine Research & Technology Symp.*, p 73–82, 2004.
- [10] K. Li, P. Hudak. Memory coherence in shared virtual memory systems. *Trans. Comp. Syst.*, 7:321–59, 1989.
- [11] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. 22nd ISCA*, p 24–36, 1995.
- [12] M. Chapman, I. Wienand, G. Heiser. Itanium page tables and TLB. Technical Report UNSW-CSE-TR-0307, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, May 2003.
- [13] L. Iftode. *Home-based Shared Virtual Memory*. Phd thesis, Princeton University, Dept of Computer Science, 1998.

Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor

Ludmila Cherkasova
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94303, USA
lucy.cherkasova@hp.com

Rob Gardner
Hewlett-Packard Laboratories
3404 E Harmony Rd., Fort Collins, CO 80528, USA
rob.gardner@hp.com

Abstract. *Virtual Machine Monitors (VMMs) are gaining popularity in enterprise environments as a software-based solution for building shared hardware infrastructures via virtualization. In this work, using the Xen VMM, we present a light weight monitoring system for measuring the CPU usage of different virtual machines including the CPU overhead in the device driver domain caused by I/O processing on behalf of a particular virtual machine. Our performance study attempts to quantify and analyze this overhead for a set of I/O intensive workloads.*

1 Introduction

The current trend toward virtualized computing resources and outsourced service delivery has caused interest to surge in *Virtual Machine Monitors (VMMs)* that enable diverse applications to run in isolated environments on a shared hardware platform. The Xen virtual machine monitor [1] allows multiple operating systems to execute concurrently on commodity x86 hardware. The recent HP Labs SoftUDC project [3] is using Xen to create isolated virtual clusters out of existing machines in a data center that may be shared across different administrative units in an enterprise. Managing this virtual IT infrastructure and adapting to changing business needs is a challenging task. In SoftUDC, virtual machines (VMs) can be migrated from one physical node to another when current physical node capacity is insufficient, or for improving the overall performance of the underlying infrastructure.

To support these management functions, we need an accurate monitoring infrastructure reporting resource usage of different VMs. The traditional monitoring system typically reports the amount of CPU allocated by the scheduler for execution of a particular VM over time. However, this method might not reveal the “true” usage of the CPU by different VMs. The reason is that virtualization of I/O devices results in an I/O model where the data transfer process involves additional system components, e.g. hypervisor and/or device driver domains. Hence, the CPU usage when the hypervisor or device driver domain handles the I/O data on behalf of the particular VM needs to be charged to the corresponding VM.

In this work, we present a lightweight, non-intrusive monitoring framework for measuring the CPU overhead in VMM related layers during I/O processing and a method for charging this overhead to VMs causing the I/O traffic. Our performance study presents measurements of the CPU overhead in the device driver domain during I/O processing and attempts to quantify and analyze the nature of this overhead.

2 Xen

Xen [1, 2] is an x86 virtual machine monitor based on a virtualization technique called *paravirtualization* [8, 1], which has been introduced to avoid the drawbacks of full virtualization by presenting a virtual machine abstraction that is similar but not identical to the underlying hardware. Xen does not require changes to the application binary interface (ABI), and hence no modifications are required to guest applications. For full details on the Xen architecture and features, we refer readers to papers [1, 2]. Here, we only touch on some implementation details of Xen that are important for our monitoring framework and performance study.

In the initial design [1], Xen itself contained device driver code and provided safe shared virtual device access. The support of a sufficiently wide variety of devices is a tremendous development effort for every OS project. In a later paper [2], the Xen team proposed a new architecture used in the latest release of Xen which allows unmodified device drivers to be hosted and executed in isolated “driver domains” which, in essence, are driver-specific virtual machines.

There is an initial domain, called *Domain0*, that is created at boot time and which is permitted to use the control interface. The control interface provides the ability to create and terminate other domains, control the CPU scheduling parameters and resource allocation policies, etc. *Domain0* also may host unmodified Linux device drivers and play the role of a driver domain. In our experimental setup, described in Section 4, we use *Domain0* as a driver domain. Devices can be shared among guest operating systems. To make this sharing work, the privileged guest hosting the device driver (e.g. *Domain0*) and the unprivileged guest domain that wishes to access the device are connected together through virtual device interfaces using device channels [2]. Xen exposes a set of clean and simple device abstractions. I/O data is transferred to and from each domain via Xen, using shared-memory, asynchronous buffer descriptor rings. In order to avoid the overhead of copying I/O data to/from the guest virtual machine, Xen implements the “page-flipping” technique, where the memory page containing the I/O data in the driver domain is exchanged with an unused page provided by the guest OS. Our monitoring framework actively exploits this feature to observe I/O communications between the guest domains and the driver domains.

3 Monitoring Framework

To implement a monitoring system that accounts for CPU usage by different guest VMs, we instrumented activity in the hypervisor CPU scheduler.

Let $Dom_0, Dom_1, \dots, Dom_k$ be virtual machines that share the host node, where Dom_0 is a privileged management domain (*Domain0*) that hosts the device drivers. Let Dom_{idle} denote a special idle domain that “executes” on the CPU when there are no other runnable domains (i.e. there is no virtual machine that is not-blocked and not-idle). Dom_{idle} is the analog to the “idle-loop” executed by an OS when there are no other runnable processes.

At any point of time, guest domain Dom_i can be in one of the following three states:

- *execution state*: domain Dom_i is currently using CPU;
- *runnable state*: domain Dom_i is not currently using CPU but is on the run queue and waiting to be scheduled for execution on the CPU;
- *blocked state*: domain Dom_i is blocked and is not on the run queue (once unblocked it is put back on the queue).

For each domain Dom_i , we collect a sequence of data describing the timing of domain state changes. Using this data, it is relatively straightforward to compute the share of CPU which was allocated to Dom_i over time.

As was mentioned in Section 2, in order to avoid the overhead of copying I/O data to/from the guest virtual machine Xen implements the “page-flipping” technique, where the memory page containing the I/O data is exchanged with an unused page provided by the guest OS. Thus, in order to account for different I/O related activities in Dom_0 (that “hosts” the unmodified device drivers), we observe the memory page exchanges between Dom_0 and Dom_i . We measure the number N_i^{mp} of memory page exchanges performed over time interval T_i when Dom_0 is in *execution state*. We derive the CPU cost (CPU time processing) of these memory page exchanges as $Cost_i^{mp} = T_i / N_i^{mp}$. After that, if there are $N_i^{Dom_i}$ memory page exchanges between Dom_0 and virtual machine Dom_i then Dom_i is “charged” for $N_i^{Dom_i} \times Cost_i^{mp}$ of CPU time processing of *Domain0*. In this way, we can partition the CPU time used by *Domain0* for processing the I/O related activities of different VMs sharing the same device driver, and “charge” the corresponding virtual machine that caused these I/O activities. Within the monitoring system, we use a time interval of 100 ms to aggregate overall CPU usage across different virtual machines.

4 Performance Study

We performed a few groups of experiments that exercise network and disk I/O traffic in order to evaluate the CPU usage caused by this traffic in *Domain0*.

All the experiments were performed on an HP x4000 Workstation with a 1.7 GHz Intel Xeon processor, 2 GB RAM, Intel e100 PRO/100 network interface, and Maxtor 40GB 7200 RPM IDE disk. For these measurements, we used the XenLinux port based on Linux 2.6.8.1 and Xen 2.0.

The first group of experiments relates to web server performance. Among the industry standard benchmarks that are used to evaluate web server performance are the SPECweb’96 [6] and SPECweb’99 [7] benchmarks. The web server performance is measured as a maximum achievable number of connections per second supported by a server when retrieving files of various sizes. Realistic web server workloads may vary significantly in both their file mix and file access pattern. The authors of an earlier study [5] established the web server performance envelope: they showed that under a workload with a short file mix the web server performance is CPU bounded, while under a workload with a long file mix the web server performance is network bounded.

To perform a sensitivity study of the CPU overhead in *Domain0* caused by different web traffic, we use Apache HTTP server version 2.0.40 running in the guest domain, and the *httperf* tool [4] for sending the client requests. The *httperf* tool provides a flexible facility for generating various HTTP workloads and for measuring server performance. In order to measure the request throughput of a web server, we invoke *httperf* on the client machine, which sends requests to the server at a fixed rate and measures the rate at which replies arrive. We run the tests with monotonically increasing request rates, until we see that the reply rate levels off and the server becomes saturated, i.e., it is operating at its full capacity. In our experiments, the http client machine and web server are connected by a 100 Mbit/s network.

We created a set of five simple web server workloads, each retrieving a fixed size file: 1 KB, 10 KB, 30 KB, 50 KB, and 70 KB. Our goal is to evaluate the CPU overhead in *Domain0* caused by these workloads. Figure 1 summarizes the results of our experiments.

Figures 1 a), b) show the overall web server performance under the studied workloads. To present all the workloads on the same scale, we show the applied load expressed as a percentage of maximum achieved throughput. For example, the maximum throughput achieved under a workload with a 1 KB file size is 900 req/s. Thus the point on the graph with X axis of 100% reveals 900 req/s throughput shown on the Y axis. Similarly, the maximum throughput achieved under a workload with 70 KB file size is 160 req/s, and this point corresponds to 100% of applied load. Figure 1 b) presents the amount of performed network I/O in KB/s as reported by the *httperf* tool. These measurements combine the HTTP requests (80 bytes long) and the HTTP responses (that include 278 bytes HTTP headers and the corresponding file as content). Figure 1 b) reveals that web server throughput is network bounded for workloads of 30-70 KB files due to network bandwidth being limited to 100Mb/s (12.5 MB/s). Another interesting feature of these workloads is apparent from Figure 1 b): the amount of transferred network I/O is practically the same for workloads of 30-70 KB files ($30KB \times 380 \text{ req/s} \approx 50KB \times 225 \text{ req/s} \approx 70KB \times 160 \text{ req/s}$).

Figure 1 c) shows the measured CPU usage by *Domain0* for each of the corresponding workloads. The CPU usage by *Domain0* increases with a higher load, reaching 24% for the workload of 1 KB files and peaking at 33-34% for workloads

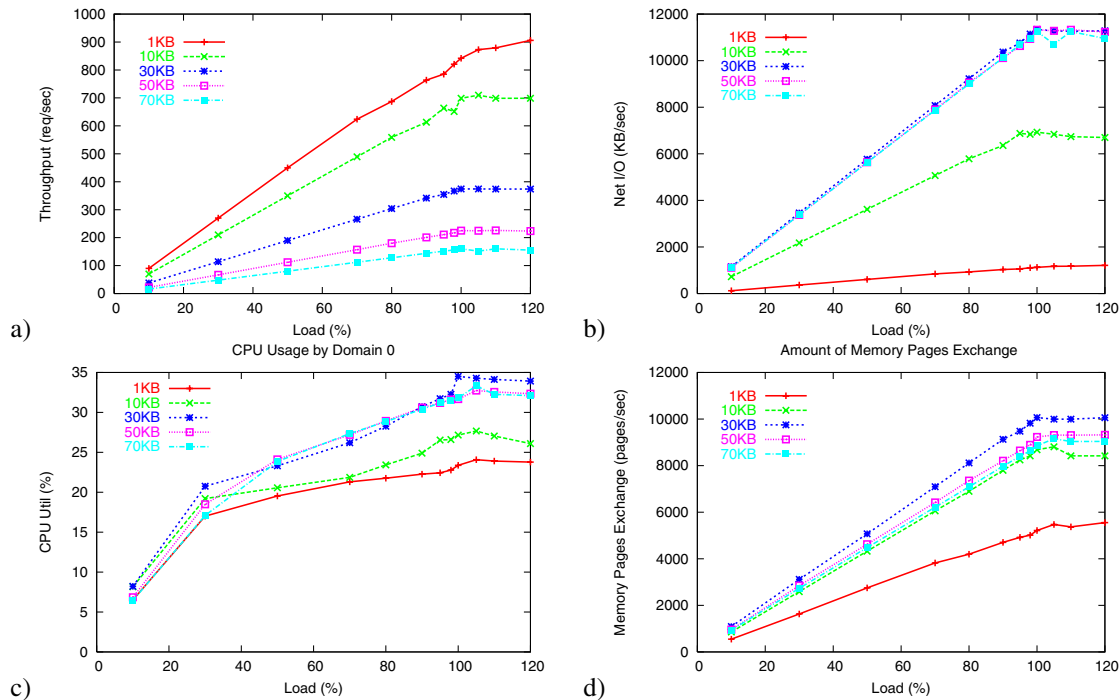


Figure 1: Summary of performance experiments with web server running in a single guest domain.

of 30-70 KB files. Measurements presented in Figure 1 c) answer one of our questions about the amount of CPU usage in *Domain0* that is caused by device driver processing for web server related workloads. The measured CPU usage presents a significant overhead, and thus should be charged to the virtual machine causing this overhead.

Our monitoring tool measures the number of memory page exchanges performed per second over time. Figure 1 d) presents the rates of memory page exchanges between *Domain0* and the corresponding guest domain. At first glance, these numbers look surprising, but under more careful analysis, they all make sense. While the memory pages are 4 KB, a single memory page corresponds to a network packet transfer whether it is a *SYN*, or *SYN-ACK*, or a packet with the HTTP request. Thus network related activities for an HTTP request/response pair for a 1 KB file require at least 5 TCP/IP packets to be transferred between the client and a web server. Thus, processing 1000 requests for a 1 KB file translates to ≈ 5000 TCP/IP packets and the corresponding rates of memory page exchanges.

Figure 2 presents the CPU usage by *Domain0* versus the amount of network I/O traffic transferred to/from a web server during the the corresponding workload. Often, the expectations are that the CPU processing overhead can be predicted from the number of transferred bytes. In the case of high volume HTTP traffic requesting small files, the small size transfers are “counter-balanced” by the high number of interrupts corresponding to the processing of the web requests. While the amount of transferred data is relatively small for the 1 KB file workload, it results in high CPU processing overhead in *Domain0* due to the vastly higher number of requests (10-20 times) for the corresponding workload as shown in Fig. 2.

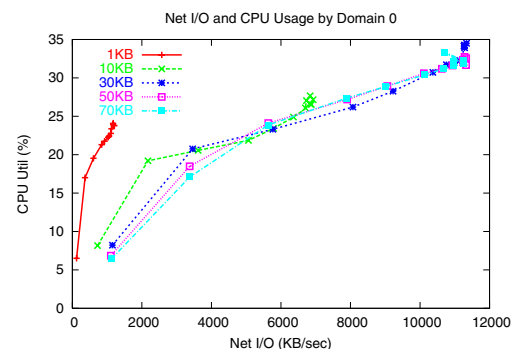


Figure 2: CPU usage by *Domain0* versus the amount of network I/O traffic transferred to/from a web server.

To complete our web server case study, we run Xen with two guest domains configured with equal resource allocations, where each guest domain runs an Apache web server. Using the *httperf* tool, we designed a new experiment where requests that are sent to a web server in *Domain1* retrieve 1 KB files, and the requests sent to a web server in *Domain2* retrieve 70 KB files. In these experiments, we use the same request rates as in a single guest domain case. Our goal is to evaluate the CPU overhead in *Domain0* caused by these workloads, as well as present the parts of this overhead attributed to *Domain1* and *Domain2*.

Figure 3 a) presents throughput achieved for both web servers under the applied workloads. It shows that a web server running in *Domain1* and serving 1 KB file workload is able to achieve 50% of the throughput achieved by a web server running in a single guest domain (see Figure 1 a). Interestingly, a web server running in *Domain2* and serving 70 KB file workload is able to achieve almost 100% of its

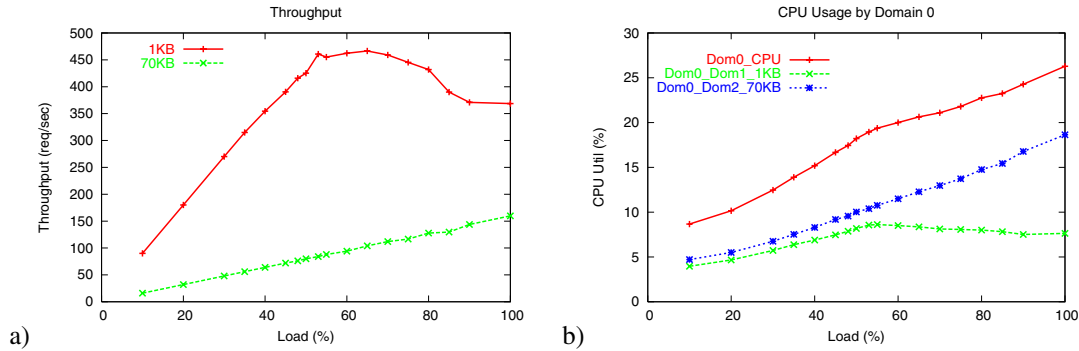


Figure 3: Summary of performance experiments with web servers running in two different guest domains.

throughput compared to the single guest domain case. This is because the performance of a web server handling a 70 KB file workload is network bounded, not CPU bounded. In the designed experiment, it can use most of the available network bandwidth, since the “competing” 1 KB file workload is CPU bounded and has network bandwidth requirements that are 70 times lower.

Figure 3 b) shows the measured CPU usage by *Domain0* and the portions attributed to *Domain1* and *Domain2*. For example, under 100% of applied load, the CPU usage by *Domain0* is 26.5%, where *Domain1* is responsible for 7.5% and *Domain2* accounts for the remaining 19% of it. Thus, it is important to capture the additional CPU overhead caused by the I/O traffic in *Domain0*, and accurately charge it to the domain causing this traffic.

The second group of experiments targets the disk I/O traffic in order to evaluate the CPU usage in *Domain0* caused by this traffic. We use the *dd* command to read 500, 1000, 5000, and 10000 blocks of size 1024 KB from the “raw” disk. Table 1 summarizes the measurements collected during these experiments. First of all, the transfer time is di-

File Size	Transfer Time (sec)	Tput (MB/s)	Domain0 CPU Usage (%)	Memory Page Exch. (pages/s)
0.5 GB	38.4 sec	13.0 MB/s	12.68%	27,342
1 GB	76.8 sec	13.0 MB/s	12.5%	27,240
5 GB	379.5 sec	13.2 MB/s	12.52%	27,508
10 GB	763.6 sec	13.1 MB/s	12.51%	27,254

Table 1: Summary of “raw” disk performance measurements.

rectly proportional to the amount of transferred data. The achieved disk bandwidth, the *Domain0* CPU usage, and the rates of memory page exchanges are consistent across different experiments. This is expected for a disk bandwidth limited workload. However, the measured rates of memory page exchanges are surprising: we expected to see around 3250 page/s (13,000 KB/s divided by 4 KB memory pages should produce ≈ 3250 page/s), but we observe rates of memory page exchanges 8 times higher. The explanation is that the block size at a “raw” disk device level is 512 bytes. Thus each 4 KB memory page is used for transferring only 512 bytes of data. This leads to rates of memory page exchanges 8 times higher and, as a result, to a significantly higher CPU overhead in *Domain0*.

We repeated the same set of experiments for a disk device that is mounted as a file system. Table 2 summarizes the results collected for the new set of experiments. The transfer

time is practically unchanged and again is directly proportional to the transferred amount of data. The achieved disk bandwidth is similar to the first set of experiments. However,

File Size	Transfer Time (sec)	Tput (MB/s)	Domain0 CPU Usage (%)	Memory Page Exch. (pages/s)
0.5 GB	38.3 sec	13.0 MB/s	4.08%	3,275
1 GB	77.8 sec	12.8 MB/s	4.1%	3,387
5 GB	386.3 sec	12.9 MB/s	3.98%	3384
10 GB	772.1 sec	13. MB/s	3.91%	3383

Table 2: Measurements for a disk device mounted as a file system.

the *Domain0* CPU usage and the rates of memory page exchanges are much lower than for the first set of experiments. The measured rates of memory page exchanges are close to our initial expectations of 3250 page/s. The measured CPU usage in *Domain0* is only about 4% for all the experiments in the second set, and it correlates well with the rates of memory page exchanges.

To quantify the overhead introduced by our instrumentation and performance monitor, we repeated all the experiments for the original Xen 2.0. The performance results for the instrumented and non-instrumented, original version of Xen 2.0 are practically indistinguishable.

In our future work, we intend to design a set of resource allocation policies that take this CPU overhead into account.

References

- [1] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. Proc. of ACM SOSP, October 2003.
- [2] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, M. Williamson. Reconstructing I/O. Tech. Report, UCAM-CL-TR-596, August 2004.
- [3] M. Kallahalla, M. Uysal, R. Swaminathan, D. Lowell, M. Wray, T. Christian, N. Edwards, C. Dalton, F. Gittler. SoftUDC: A Software-based data center for utility computing. IEEE Computer, Special issue on Internet Data Centers, pp. 46-54, November, 2004.
- [4] D. Mosberger, T. Jin. Httpperf—A Tool for Measuring Web Server Performance. Proc. of Workshop on Internet Server Performance, 1998.
- [5] F. Prefect, L. Doan, S. Gold, and W. Wilcke. Performance Limiting Factors in Http (Web) Server Operations. Proc. of COMPCON’96, Santa Clara, 1996, pp.267-273.
- [6] The Workload for the SPECweb96 Benchmark. <http://www.specbench.org/osg/web96/workload.html>
- [7] The Workload for the SPECweb99 Benchmark. <http://www.specbench.org/osg/web99/workload.html>
- [8] A. Whitaker, M. Shaw, and S. Gribble. Denali: A Scalable Isolation Kernel. Proc. of ACM SIGOPS European Workshop, September 2002.

Fast Transparent Migration for Virtual Machines

Michael Nelson, Beng-Hong Lim, and Greg Hutchins

*VMware, Inc.
Palo Alto, CA 94304*

Abstract

This paper describes the design and implementation of a system that uses virtual machine technology [1] to provide fast, transparent application migration. This is the first system that can migrate unmodified applications on unmodified mainstream Intel x86-based operating system, including Microsoft Windows, Linux, Novell NetWare and others. Neither the application nor any clients communicating with the application can tell that the application has been migrated. Experimental measurements show that for a variety of workloads, application downtime caused by migration is less than a second.

1 Introduction

Fast transparent migration can improve global system utilization by load balancing across physical machines, and can improve system serviceability and availability by moving applications off machines that need servicing or upgrades. This paper describes a migration system, named *VMotion*, that has been shipping since 2003 as an integral part of the VMware VirtualCenter product [2]. Future VMware products will utilize *VMotion* to automate load balancing across large numbers of servers.

This paper makes the following contributions.

- It describes the first system to provide transparent virtual machine migration of existing applications and operating systems; neither the applications nor the operating systems need to be modified.
- It is the first paper to provide performance measurements of hundreds of virtual machine migrations of concurrently running virtual machines with standard industry benchmarks.
- It characterizes the overheads and resources required during virtual machine migration.

2 Virtual Machine Migration

Virtual machine migration takes a running virtual machine and moves it from one physical machine to another. This migration must be transparent to the guest operating system, applications running on the operating system, and remote clients of the virtual machine. It should appear to all parties involved that the virtual machine did not change its location. The

only perceived change should be a brief slowdown during the migration and a possible improvement in performance after the migration because the VM was moved to a machine with more available resources.

The migration system presented in this paper is part of the VMware VirtualCenter product that manages VMware ESX Server [3]. VMware ESX Server consists of two main components that implement the virtual platform: the virtual machine monitor (VMM) and the vmkernel. A guest operating system such as Windows or Linux runs on top of this virtual platform (see Figure 1). The VMM handles the execution of all instructions on the virtual CPU and the emulation of all virtual devices. The vmkernel schedules the VMM for each virtual machine and allocates and manages the resources needed by the virtual machines.

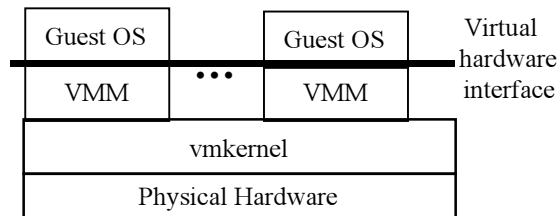


Figure 1. VM platform layers in VMware ESX Server.

Virtual machines provide a natural platform for migration by encapsulating all of the state of the hardware and software running within the virtual machine. There are three kinds of state that need to be dealt with when migrating a VM:

- 1) The virtual device state including the state of the CPU, the motherboard, networking and storage adapters, floppy disks, and graphics adapters.
- 2) External connections with devices including networking, USB devices, SCSI storage devices, and removable media such as CD-ROMs.
- 3) The VM's physical memory.

The actual migration process involves several steps:

- 1) Initiating the migration by selecting the VM to migrate and its destination.
- 2) Pre-copying the memory state of the VM to the destination while the VM is running on the source.

- 3) Quiescing the VM and sending the non-memory state.
- 4) Transferring control of the VM to the destination and resuming it at the destination.
- 5) Sending any remaining memory state and removing the dependency on the source machine.

The remainder of this section describes the steps involved in migrating three of the most important components of a VM: networking, SCSI storage devices, and physical memory.

Networking In order for a migration to be transparent all network connections that were open before a migration must remain open after the migration completes. The VMware ESX Server virtual networking architecture makes this possible.

A virtual Ethernet network interface card (VNIC) is provided as part of the virtual platform. Like a physical NIC, the VNIC has a MAC address that uniquely identifies it on the local network. Each VNIC is associated with one or more physical NICs that are managed by the vmkernel. The VNICs of many VMs can be attached to the same physical NIC.

Since each VNIC has its own MAC address that is independent of the physical NIC's MAC address, virtual machines can be moved while they are running between machines and still keep network connections alive as long as the new machine is attached to the same subnet as the original machine.

SCSI Storage We rely on storage area networks (SAN) or NAS to allow us to migrate connections to SCSI devices. We assume that all physical machines involved in a migration are attached to the same SAN or NAS server. This allows us to migrate a SCSI disk by reconnecting to the disk on the destination machine.

Physical Memory The physical memory of the virtual machine is the largest piece of state that needs to be migrated. Pausing a VM while the entire memory state is transferred will result in the VM being inaccessible for too long. We address this problem by copying the physical memory state from the source machine to the destination machine while the VM is running. This is possible because of the way that we manage the physical memory of the VM [3].

Each virtual machine expects to have a fixed set of physical address ranges that map to physical memory. VMware ESX Server dynamically allocates the real machine's physical memory to the running virtual machines. This requires adding a level of indirection to provide the physical memory layout expected by a

guest operating system. All direct accesses to the VM's physical memory, as well as all writes to memory mapping hardware and page tables, are intercepted by the VMM. The VMM then translates these physical addresses into the actual machine addresses. Once the VM's memory mapping hardware and page tables are properly set up, the VM can run without any additional physical-to-machine address translation overhead.

We use this level of indirection to iteratively pre-copy the memory [4] while the VM continues to run on the source machine. The first step copies over the entire physical memory of the VM. Before each page is copied, it is marked read-only so that any modifications to the page can be detected by the VMM. When the first step is completed, some memory will have been modified by the still-running VM. The modified pages are then copied again to the destination while the VM continues to run. This procedure is then repeated until either the number of modified pages is small enough or there is insufficient forward progress. Currently we terminate the pre-copy when there are less than 16 megabytes of modified pages left or there is a reduction in changed pages of less than 1 megabyte.

3 Performance Measurements

This section investigates the performance characteristics of the virtual machine migration scheme described above. It presents measurements of the time to migrate a VM and the period during which the VM is unavailable. It also characterizes the effect of CPU resource allocation. Most importantly, it shows that for a variety of workloads VM migration can be fast and transparent to applications and operating systems.

3.1 Experimental Setup

All experiments were performed on a pair of identical Dell 1600SC servers each with two 2.4GHz Intel Xeon processors and 1 GB of RAM. The servers are connected to an EMC CLARiiON SAN via Qlogic 2300 HBAs. Intel Pro/1000 Gbit NICs are used to transfer the state of the VMs.

Each experiment migrates a single VM 50 times between two servers with 5-second intervals. The numbers reported are the average of the 50 migrations.

The experiments use the following VM workloads:

- **idle**: An idle Windows 2000 Server.
- **kernel-compile**: Linux kernel compilation in RedHat 7.2.
- **iometer**: Iometer [7] running on Windows 2000 Server. Iometer was configured to run disk I/O

with 3 worker threads each performing I/O to a 500MB file with up to five outstanding writes.

- **memtest86**: Memtest86 [5], which continuously reads and writes memory, running test #1 in a loop
- **dbhammer**: Database Hammer [6], a client/server database load generator running on Windows 2000 Server. The server was migrated while the client was running on a third physical machine.

Except for the measurements in Section 3.4, the VM being migrated was the only VM running on the source machine and there were no VMs running on the destination machine.

3.2 Migration Time

Migration proceeds in several distinct steps. We are most interested in the downtime during which the VM is unavailable. This period must be short enough to avoid any noticeable loss of service from the VM. We are also interested in the total end-to-end time of a migration during which machine resources are consumed to perform the migration.

Downtime The total downtime consists of the time necessary to quiesce the VM on the source, transfer the device state to the destination, load the device state, and copy over all the remaining memory pages concurrently with loading the device state.

Figure 2 shows that the total downtime is less than one second for all workloads except *memtest86* and rises minimally with increasing memory sizes. *memtest86* is a pathological case where all the memory was modified during the pre-copy so that the VM downtime equals the time necessary to send the VM's entire memory.

End-to-end Time Figure 3 shows that the total end-to-end time depends strongly on the size of the VM's

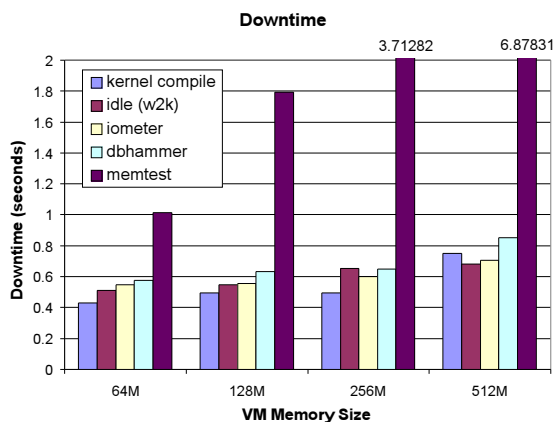


Figure 2: Downtime during migration for various workloads and VM memory sizes.

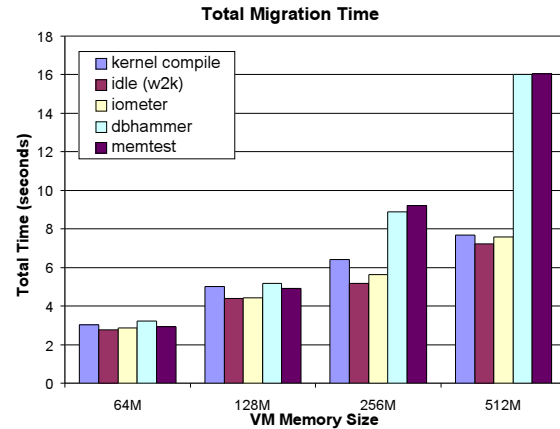


Figure 3. Total end-to-end time for a migration.

memory, and confirms the need to keep the VM running during most of this time. With pre-copying, the VM continues to run while memory is being transferred to the destination.

The number of pre-copy iterations required to migrate each workload was small. All workloads except for *memtest86* took 1 or 2 rounds before the number of modified pages was small enough to terminate the pre-copy. It took 2 or 3 rounds before the pre-copy was aborted because of lack of progress for *memtest86*.

3.3 Effect of Pre-copy

Figure 4 shows the effect of pre-copying memory on network throughput as measured by the *dbhammer* client during a window of three back-to-back migrations of the *dbhammer* server. The three large drops in throughput correspond to the downtime.

The smaller 20% drops in throughput are caused by the

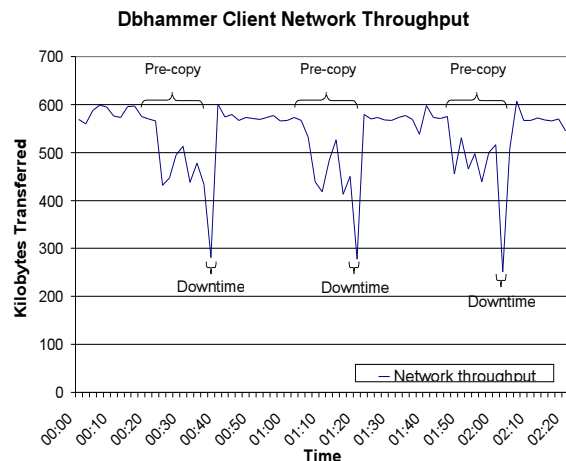


Figure 4. Effect on network throughput as seen by the client of a migrating *dbhammer* server.

pre-copy. This drop is caused by the overhead of marking the pre-copied pages as read-only, which involves halting all virtual CPUs, and the overhead of handling any protection faults to the read-only pages.

3.4 Resource Management

To monitor the effect of resource management, the source physical machine was loaded with the equivalent of 20 CPU-bound virtual machines, and the time to migrate an idle 512MB Windows 2000 Server VM was measured under different resource reservations. Figure 5 shows that reserving 30% of a CPU for migration minimizes the pre-copy time. This implies that it takes around 30% of a CPU to attain the maximum network throughput over the gigabit link.

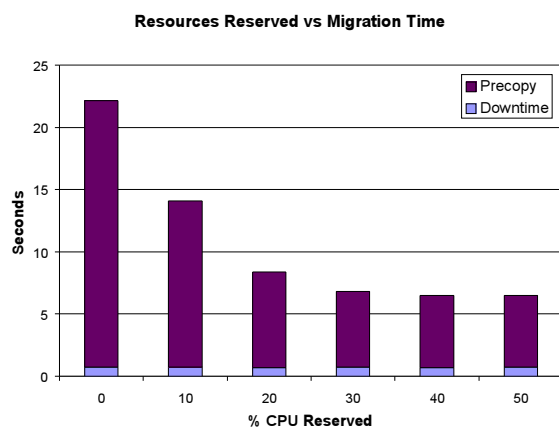


Figure 5. Effect of CPU reservation on migration from a heavily loaded source machine.

Even though the pre-copy time increased when insufficient CPU was reserved for the migration, the downtime remains small regardless of the amount of reserved CPU. It requires little CPU time to quiesce the VM and transfer over the non-memory state.

4 Related Work

There is large amount of previous work done on transparently migrating processes. Zap [8] is a recent system that provides process migration and contains a good discussion of previous work in this area.

The only other system that migrates virtual machines is described by Hansen and Jul in [9]. The fundamental difference between their work and ours is that they require guest OS modifications in order to perform the migration. Whereas our system can migrate any OS that runs on the Intel x86 architecture including closed-source operating systems such as Microsoft Windows, their system can only migrate operating systems that can be modified to work in their environment.

5 Conclusions

Previous attempts at application migration have had limited success primarily because of the difficulty of encapsulating the state of a running application. Virtual machines solve this problem by allowing not only an application to be encapsulated, but the operating system and the hardware as well. We have described a migration implementation that allows an entire running VM to be migrated from one physical machine to another. The migration is completely transparent to the application, the operating system and remote clients.

The method for migrating the physical memory of a VM is critical to providing transparent migration. It takes many seconds, even over fast networks, and significant CPU resources to transfer large memories. We have shown that transferring the memory while a VM is running minimizes the downtime. Our measurements show that a VM normally experiences less than one second of down time. Also, the end-to-end time of the migration and the impact on other VMs running on the machines involved in the migration can be controlled by properly managing CPU resources.

References

1. R. Goldberg. "Survey of Virtual Machine Research," *IEEE Computer*, 7(6), June 1974.
2. "Building Virtual Infrastructure with VMware Virtual Center," http://www.vmware.com/pdf/vi_wp.pdf.
3. C. Waldspurger. "Memory Resource Management in VMware ESX Server," *Proc. Of the 5th Operating Systems Design and Implementation*, December 2002.
4. M. Theimer, K. Lantz, and D. Cheriton. "Preemptable Remote Execution Facilities for the V-System," *Proc. Of the 10th Symposium on Operating System Principles*, December 1985.
5. "Memtest86 - A Stand-alone Memory Diagnostic," <http://www.memtest86.com/>.
6. "Microsoft SQL Server: Resource Kit," <http://www.microsoft.com/sql/techno/reskit>.
7. "Iometer Project," <http://www.iometer.org/>.
8. S. Osman, et al. "The Design and Implementation of Zap: A System for Migrating Computing Environments," *Proc. Of the 5th Operating Systems Design and Implementation*, December 2002.
9. J.G. Hansen and E. Jul, "Self-migration of Operating Systems," *Proc. Of the 11th ACM European SIGOPS Workshop*, September 2004.

Performance of Multithreaded Chip Multiprocessors And Implications For Operating System Design

Alexandra Fedorova^{†‡}, Margo Seltzer[†], Christopher Small[‡] and Daniel Nussbaum[‡]
[†]*Harvard University*, [‡]*Sun Microsystems*

ABSTRACT

We investigated how operating system design should be adapted for multithreaded chip multiprocessors (CMT) – a new generation of processors that exploit thread-level parallelism to mask the memory latency in modern workloads. We determined that the L2 cache is a critical shared resource on CMT and that an insufficient amount of L2 cache can undermine the ability to hide memory latency on these processors. To use the L2 cache as efficiently as possible, we propose an L2-conscious scheduling algorithm and quantify its performance potential. Using this algorithm it is possible to reduce miss ratios in the L2 cache by 25-37% and improve processor throughput by 27-45%.

1. INTRODUCTION

This paper explores the subject of operating system design for multithreaded chip multiprocessors (CMT). CMT processors combine chip multiprocessing (CMP) and hardware multithreading (MT). A CMP processor includes multiple processor cores on a single chip, which allows more than one thread to be active at a time and improves utilization of chip resources. An MT processor interleaves execution of instructions from different threads. As a result, if one thread blocks on a memory access or some other long operation, other threads can make forward progress. Numerous studies have demonstrated the performance benefits of CMP and MT [1-4, 8, 15, 16, 20].

The hardware industry is turning to CMT as a way to improve future application performance because conventional techniques for hiding the latency of long operations, such as branch prediction and out-of-order execution, are failing to work for modern applications. These applications, such as web services, application servers, and on-line transaction processing systems, usually include multiple threads of control executing short sequences of integer operations, with frequent dynamic branches. This structure decreases cache locality and branch prediction accuracy and causes frequent processor stalls [7, 17, 18, 19]. Add to that the growing gap between processor and memory performance, and the result is very poor processor

pipeline utilization: even relatively simple SPEC CPU benchmarks have pipeline utilizations as low as 19% [9]. This means that the majority of the time, the processor pipeline is unused. CMT processors address this problem. Most of the new processors released by IBM, Intel and Sun Microsystems today are MT, CMP or CMT [5, 6, 12].

CMTs can be equipped with dozens of simultaneously active thread contexts (e.g., Sun's Niagara processor will have eight cores, each with four thread contexts [5]), and, as a result the competition for shared resources is intense. Our approach is to identify the shared resource that is likely to become the performance bottleneck and then investigate operating system approaches for improving its efficiency.

In this paper we focus on processor caches¹. We have found that the latency resulting from poor hit rates in the L1 cache can be effectively hidden by hardware multithreading, but that high contention for the L2 can significantly hurt overall processor performance. This result drove us to investigate how much potential there is in using OS scheduling to improve L2 (and subsequently overall processor) performance.

We have designed an OS scheduling algorithm based on the balance-set principle [14], and found that it has the potential to reduce the L2 cache miss ratios by 25-37%, yielding a performance improvement of 27-45% – an improvement that could have been achieved in hardware only by doubling the size of the L2 cache.

2. PERFORMANCE BOTTLENECKS

In this section we show that while hardware multithreading does an excellent job of hiding latency resulting from L1 cache misses, its ability to hide memory latency resulting from L2 cache misses is limited.

We performed experiments using a CMT processor simulator built on top of Simics [10, 11]. We simulate a chip multiprocessor where each multithreaded core has a simple RISC pipeline supporting simultaneous execution of four threads, shared instruction and data caches (16KB and 8KB respectively), and a shared TLB. There is a unified shared L2 cache per chip, whose size we vary

depending on the experiment. For more details on our simulator and for the explanation of our choices for system configuration parameters, please refer to our earlier papers [11, 21].

To analyze processor sensitivity to the L1 cache miss ratio, we measured cache miss ratios and instructions per cycle (IPC) for the benchmarks from the SPEC CPU2000 suite, varying the size of the data cache from 8KB to 128KB. In each experiment we ran four copies of the same benchmark on a single-core machine – the threads running the workload shared the core's data cache.

Figure 1 shows the average miss ratios and IPC for the benchmarks. The key point of this figure is that even though the increase of cache miss ratios is significant (from 8% for the 128KB cache to 25% for the 8KB cache), the IPC is relatively insensitive to this variation.

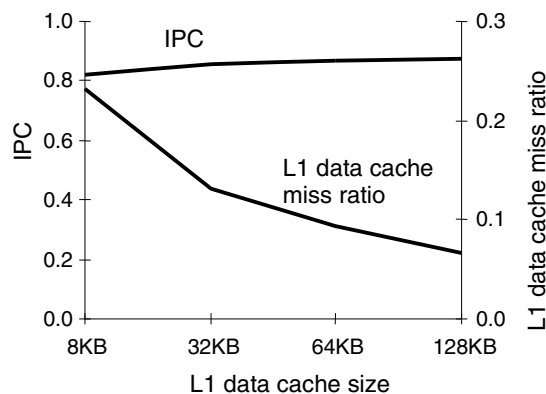


Figure 1. IPC and L1 data cache miss ratio for the SPEC workload

Now, consider a similar experiment with the L2 cache. We configured our simulator to have two cores, and ran the SPEC benchmarks simultaneously, creating a workload of heterogeneous threads.

Figure 2 shows the processor IPC and the L2 cache miss ratio for various L2 cache sizes. IPC degradation is evident as the L2 cache becomes smaller – this shows that the processor performance is significantly dependent upon the L2 cache.

This result is worthy of serious attention. Modern applications exhibit a trend of becoming progressively more memory-intensive. While CMT processors may be equipped with enough cache for the time being, it is likely that in the future it will be more difficult for CMT processors to satisfy application cache needs. Equipping these processors with larger L2 caches may be difficult: as the microchip technology is moving beyond the 90-nm mark, packing more and more transistors on a processor becomes increasingly more

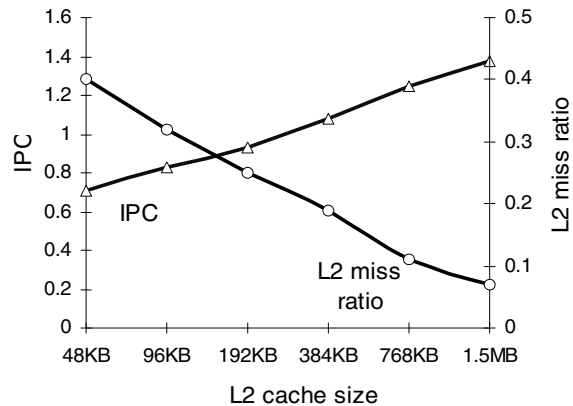


Figure 2. IPC and L2 miss ratio for the SPEC workload

complicated due to limitations of silicon technology. To ensure that our systems run well in the future, it is important that the software community develop techniques to improve resource utilization of CMT processors. In the next section we describe an L2-cache-conscious scheduling algorithm and quantify the potential performance improvement that it can produce.

3. BALANCE-SET SCHEDULING

Balance-set scheduling was proposed by Denning [14] as a way to improve the performance of virtual memory. We evaluated the effectiveness of this approach for the L2 cache. The idea behind balance-set scheduling is as follows. Separate all runnable threads into subsets, or groups, such that the combined working set of each group fits in the cache. Then, schedule a group at a time for the duration of the scheduling time slice. By making sure that the working set of each scheduled group fits in the cache, this algorithm reduces cache miss ratios.

However, we found that working set size is not a good indicator of a workload's cache behavior [21]: the reuse pattern of memory locations in the working set is more important than the size of the working set. In order to estimate a cache miss ratio produced by a group of threads we used a cache model for single-threaded workloads developed by Berg and Hagersten [13], and adapted it to work for multi-threaded workloads [21].

Using this model we are able to estimate cache miss ratios of multithreaded workloads to within 17% of their actual values, on average. Such accuracy is satisfactory, because it is sufficient to distinguish between those workloads that fit in the cache and those that thrash.

Once we are able to estimate the cache miss ratio of any group of threads we can decide which threads should be scheduled together. By scheduling threads in groups that have low cache miss ratios, we ensure that

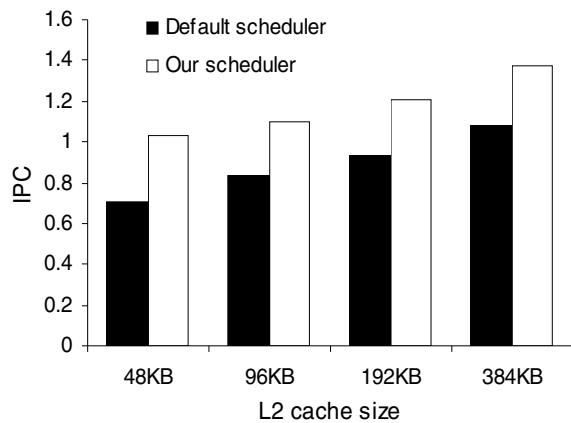


Figure 3. IPC achieved with the default scheduler and the balance-set scheduler.

the miss ratio is always kept low, and the overall IPC is high. We also need to make sure that each runnable thread is included in at least one of the scheduled groups, so that none are starved.

The steps involved in our algorithm are:

1. Estimate cache miss ratios for all possible groups of threads using the Berg-Hagersten-based model.
2. Select cache miss ratio threshold
3. Schedule those thread groups whose estimated cache miss ratio is below the threshold.

Before implementing this algorithm in the operating system, we wanted to be sure that it would be worth the effort. Therefore, we quantified the potential for performance improvement using a scheduler prototype, which we implemented partially at user-level and partially inside the simulator. We now briefly describe the prototype implementation of these steps; a more detailed description appears in our earlier work [21].

Estimating cache miss ratios using the Berg-Hagersten model requires monitoring threads' memory re-use patterns. To implement such monitoring it is necessary to construct a sample of memory locations that a thread references and then to watch how often those locations are reused. Using this approach in a real system is expensive, because it requires handling frequent processor traps. Although we are working on a low-overhead way to approximate the measurements produced by the memory-monitoring approach, in this analysis we use the memory-monitoring approach because it provides the best accuracy. Our hardware simulator analyzes memory-reuse patterns of threads, and produces the data that we use to estimate cache-miss ratios for all groups of threads.

The cache miss ratio threshold guides the scheduling decisions. Only the thread groups whose

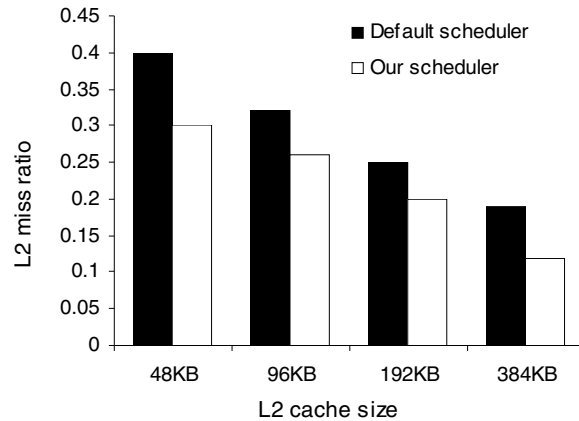


Figure 4. L2 cache miss ratios achieved with the default scheduler and the balance-set scheduler.

estimated miss ratio is below the threshold will be scheduled. We select the cache-miss ratio threshold such that each thread is included in at least one of the groups that satisfies the threshold; of all such possible threshold values, we choose the lowest. This way we ensure that none of the threads are starved.

In our prototype we enforced scheduling decisions at user-level, binding threads to processors using the mechanisms available through a user-level library on the Solaris™ operating system.

We now present performance results achieved using our algorithm. We use the multi-process SPEC CPU workload – the same that we used for the analysis of IPC sensitivity in the previous section. We compare the IPC and the L2 cache miss ratio of this workload achieved using the default Solaris™ scheduler, and using our balance-set scheduler prototype. Figure 3 presents the IPC for both schedulers, Figure 4 presents the L2 cache miss ratio. In all cases, the balance-set scheduler outperforms the default scheduler. The IPC gain from the balance-set scheduler is from 27% (384KB L2) to 45% (48KB L2). This improvement in the IPC is due to reduction in cache miss ratios. As Figure 4 shows, with balance-set scheduling we were able to reduce the L2 miss ratios by 25-37%.

The performance improvement resulted from constructing thread groups so that they would share the cache amiably, producing a low L2 miss ratio, and achieving high IPC as a result.

4. DISCUSSION

The magnitude of the performance improvement from balance-set scheduling depends on the properties of the workload. Since the balance-set scheduler exploits the disparity in threads' cache behaviors, if all threads in the workload behave in a similar manner,

large performance gains cannot be realized. Understanding how the characteristics of the workload affect potential performance gains is the subject of future work. We also need to determine how balance-set scheduling affects fairness: sacrificing fairness for performance is a canonical trade-off made in scheduling algorithms.

Implementing cache-miss ratio analysis using memory monitoring and examining all possible groups of threads to select those that satisfy the miss ratio threshold is expensive. We are working on a novel low-overhead way of approximating these operations and are now implementing the balance-set scheduler in Solaris™ 10.

5. CONCLUSIONS

In this paper we presented results of the first study evaluating the performance of a CMT processor. We analyzed how contention for L1 and L2 caches affects performance. We determined that contention for the L2 cache has the greatest effect on system performance – therefore, this is where system designers should focus their optimization efforts.

We investigated how to leverage the operating system scheduler to reduce the pressure on the L2 cache, using balance-set scheduling.

We demonstrated that with balance-set scheduling it is possible to reduce the L2 cache miss ratio by 25-37% and increase performance by 27-45%.

The implementation of the scheduler is currently under way. In the future, we also plan to investigate how workload characteristics affect the potential performance gains from this algorithm and the associated fairness tradeoffs.

6. ACKNOWLEDGEMENTS

The authors thank Tim Hill, John Davis and James Laudon of Sun Microsystems, and Eric Berg of University of Uppsala for help with preparation of this paper.

7. REFERENCES

- [1] R. Alverson et al. The Tera Computer System. *Proc. 1990 Intl. Conf. on Supercomputing*.
- [2] A. Agarwal, B-H. Lim, D. Kranz, J. Kubiawicz. APRIL: A Processor Architecture for Multiprocessing. *ISCA*, June 1990.
- [3] J. Laudon, A. Gupta, M. Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. *ASPLOS VI*, October 1994.
- [4] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, D. Tullsen. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM TOCS 15*, 2, August 1997.
- [5] Jonathan Schwartz on Sun's Niagara processor: http://blogs.sun.com/roller/page/jonathan/20040910#the_difference_between_humans_and
- [6] Intel web site. <http://www.intel.com/pressroom/archive/speeches/otellini20030916.htm>
- [7] J. Lo et al. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. *ISCA*, June 1998.
- [8] N. Tuck, D. Tullsen, Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. *PACT*, September 2003.
- [9] D. Tullsen, S. Eggers, H. Levy, Simultaneous Multithreading: Maximizing On-Chip Parallelism. *ISCA*, June 1995.
- [10] P. Magnusson et al. SimICS/sun4m: A Virtual Workstation. *USENIX*, June 1998.
- [11] D. Nussbaum, A. Fedorova, C. Small, The Sam CMT Simulator Kit. *Sun Microsystems TR 2004-133*, March 2004.
- [12] IBM eServer iSeries Announcement. <http://www-1.ibm.com/servers/eserver/iseries/announce/>
- [13] E. Berg, E. Hagersten. Efficient Data-Locality Analysis of Long-Running Applications. TR 2004-021, University of Uppsala, May 2004
- [14] P. Denning. Thrashing: Its causes and prevention. *Proc. AFIPS 1968 Fall Joint Computer Conference*, 33, pp. 915-922, 1968.
- [15] R. Eickenmeyer et al. Evaluation of multithreaded uniprocessors for commercial application environments. *ISCA '96*.
- [16] J. M. Borkenhagen, R. J Eickemeyer, R. N. Kalla, S.R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development* 44, 6, pp. 885.
- [17] A. Ailamaki, D. DeWitt, M. Hill, D. Wood. DBMSs on modern processors: Where does time go? *VLDB '99*, September 1999.
- [18] A. Barroso, K. Gharachorloo, E. Bugnion. Memory System Characterization of Commercial Workloads. *ISCA '98*.
- [19] K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker. Performance characterization of a Quad Pentium Pro SMP using OLTP Workloads. *ISCA '98*.
- [20] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson and K. Chang. The Case for a Single-Chip Multiprocessor. *ASPLOS* 1996.
- [21] A. Fedorova, M. Seltzer, C. Small and D. Nussbaum. Throughput-Oriented Scheduling On Chip Multithreading Systems. *Technical Report TR-17-04*, Harvard University, August 2004.

ⁱ We found previously that other shared resources have a smaller potential of becoming performance bottlenecks [21].

Hyper-Threading Aware Process Scheduling Heuristics

James R. Bulpin and Ian A. Pratt

University of Cambridge Computer Laboratory

James.Bulpin@cl.cam.ac.uk, <http://www.cl.cam.ac.uk/netos>

Abstract

Intel Corporation's "Hyper-Threading Technology" is the first commercial implementation of simultaneous multithreading. Hyper-Threading allows a single physical processor to execute two heavyweight threads (processes) at the same time, dynamically sharing processor resources. This dynamic sharing of resources, particularly caches, causes a wide variety of inter-thread behaviour. Threads competing for the same resource can experience a low combined throughput.

Hyper-Threads are abstracted by the hardware as logical processors. Current generation operating systems are aware of the logical-physical processor hierarchy and are able to perform simple load-balancing. However, the particular resource requirements of the individual threads are not taken into account and sub-optimal schedules can arise and remain undetected.

We present a method to incorporate knowledge of per-thread Hyper-Threading performance into a commodity scheduler through the use of hardware performance counters and the modification of dynamic priority.

1 Introduction

Simultaneous multithreaded (SMT) processors allow multiple threads to execute in parallel, with instructions from multiple threads able to be executed during the same cycle [10]. The availability of a large number of instructions increases the utilization of the processor because of the increased instruction-level parallelism.

Intel Corporation introduced the first commercially available implementation of SMT to the Pentium 4 [2] processor as "Hyper-Threading Technology" [3, 5, 4]. Hyper-Threading is now common in server and desktop Pentium 4 processors and becoming available in the mobile version. The individual Hyper-Threads of a physical processor are presented to the operating system as logical processors. Each logical processor can execute

a heavyweight thread (process); the OS and applications need not be aware that the logical processors are sharing physical resources. However, some OS awareness of the processor hierarchy is desirable in order to avoid circumstances such as a two physical processor system having two runnable processes scheduled on the two logical processors of one package (and therefore sharing resources) while the other package remains idle. Current generation OSs such as Linux (version 2.6 and later versions of 2.4) and Windows XP have this awareness.

When processes share a physical processor the sharing of resources, including the fetch and issue bandwidth, means that they both run slower than they would do if they had exclusive use of the processor. In most cases the combined throughput of the processes is greater than the throughput of either one of them running exclusively — the system provides increased system-throughput at the expense of individual processes' throughput. The system-throughput "speedup" of running tasks using Hyper-Threading compared to running them sequentially is of the order of 20% [1, 9]. We have shown previously that there are a number of pathological combinations of workloads that can give a poor system-throughput speedup or give a biased per-process throughput [1]. We argue that the operating system process scheduler can improve throughput by trying to schedule processes simultaneously that have a good combined throughput. We use measurement of the processor to inform the scheduler of the realized performance.

2 Performance Estimation

In order to provide throughput-aware scheduling the OS needs to be able to quantify the current per-thread and system-wide throughput. It is not sufficient to measure throughput as instructions per cycle (IPC) because processes with natively low IPC would be misrepresented. We choose instead to express the throughput of a process as a *performance ratio* specified as its rate of execution under Hyper-Threading versus its rate of execution

when given exclusive use of the processor. An application that takes 60 seconds to execute in the exclusive mode and 100 seconds when running with another application under Hyper-Threading has a performance ratio of 0.6. The *system speedup* of a pair of simultaneously executing processes is defined as the sum of their performance ratios. Two instances of the previous example running together would have a system speedup of 1.2 — the 20% Hyper-Threading speedup described above.

The performance ratio and system speedup metrics both require knowledge of a process' exclusive mode execution time and are based on the complete execution of the process. In a running system the former is not known and the latter can only be known once the process has terminated by which time the knowledge is of little use. It is desirable to be able to estimate the performance ratio of a process while it is running. We want to be able to do this online using data from the processor hardware performance counters. A possible method is to look for a correlation between performance counter values and calculated performance; work on this estimation technique is ongoing, however, we present here a method used to derive a model for online performance ratio estimation using an analysis of a training workload set.

Using a similar technique to our previous measurement work [1] we executed pairs of SPEC CPU2000 benchmark applications on the two logical processors of a Hyper-Threaded processor; each application running in an infinite loop on its logical processor. Performance counter samples were taken at 100ms intervals with the counters configured to record for each logical processor the cache miss rates for the L1 data, trace- and L2 caches, instructions retired and floating-point operations. A stand-alone base dataset was generated by executing the benchmark applications with exclusive use of the processor, recording the number of instructions retired over time. Execution runs were split into 100 windows of equal instruction count. For each window the number of processor cycles taken to execute that window under both Hyper-Threading and exclusive mode were used to compute a performance ratio for that window. The performance counter samples from the Hyper-Threaded runs were interpolated and divided by the cycle counts to give events-per-cycle (EPC) data for each window. A set of 8 benchmark applications (integer and floating-point, covering a range of behaviours) were run in a cross-product with 3 runs of each pair, leading to a total of 16,800 windows each with EPC data for the events for both the application's own, and the "background" logical processor. A multiple linear regression analysis was performed using the EPC data as the explanatory variables and the application's performance ratio as the dependent variable. The coefficient of determination (the R^2 value), an indication of how much of the variability of the dependent

variable can be explained by the explanatory variables, was 66.5%, a reasonably good correlation considering this estimate is only to be used as a heuristic. The coefficients for the explanatory variables are shown in Table 1 along with the mean EPC for each variable (shown to put the magnitudes of the variables into context). The fourth column of the table indicates the importance of each counter in the model by multiplying the standard deviation of that metric by the coefficient; a higher absolute value here shows a counter that has a greater effect on the predicted performance. Calculation of the *p-values* showed the L2 miss rate of the background process to be statistically insignificant (in practical terms this metric is covered largely by the IPC of the background process).

The experiment was repeated with a different subset of benchmark applications and the MLR model was used to predict the performance ratio for each window. The coefficient of correlation between the estimated and measured values was 0.853, a reasonably strong correlation.

We are investigating refinements to this model by considering other performance counters and input data.

3 Scheduler Modifications

Rather than design a Hyper-Threading aware scheduler from the ground up, we argue that gains can be made by making modifications to existing scheduler designs. We wish to keep existing functionality such as starvation avoidance, static priorities and (physical) processor affinity. A suitable location for Hyper-Threading awareness would be in the calculation of dynamic priority; a candidate runnable process could be given a higher dynamic priority if it is likely to perform well with the process currently executing on the other logical processor.

Our implementation uses the Linux 2.4.19 kernel.

Counter	Coefficient (to 3 S.F.)	Mean events per 1000 Cycles (to 3 S.F.)	Importance (coeff. x st.dev.)
(Constant)	0.4010		
TC-subj	29.7000	0.554	26.2
L2-subj	55.7000	1.440	87.2
FP-subj	0.3520	52.0	29.8
Insts-subj	-0.0220	258	-4.3
L1-subj	2.1900	10.7	15.4
TC-back	32.7000	0.561	29.0
L2-back	1.5200	1.43	2.3
FP-back	-0.4180	52.6	-35.3
Insts-back	0.5060	256	99.7
L1-back	-3.5400	10.6	-25.3

Table 1: Multiple linear regression coefficients for estimating the performance ratio of the subject process. The performance counters for the logical processor executing the subject process are suffixed "subj" and those for the background process's logical processor, "back".

This kernel has basic Hyper-Threading support in areas other than the scheduler. We modify the `goodness()` function which is used to calculate the dynamic priority for each runnable task when a scheduling decision is being made; the task with the highest goodness is executed. We present two algorithms: “*tryhard*” which biases the goodness of a candidate process by how well it has performed previously when running with the process on the other logical processor, and “*plan*” which uses a user-space tool to process performance ratio data and produce a scheduling plan to be implemented (as closely as possible) in a gang-scheduling manner.

For both algorithms the kernel keeps a record of the estimated system-speedups of pairs of processes. The current *tryhard* implementation uses a small hash table based on the process identifiers (PIDs). The performance counter model described above is used for the estimates. The goodness modification is to lookup the recorded estimate for the pair of PIDs of the candidate process and the process currently running on the other logical processor.

For each process *p*, *plan* records the three other processes that have given the highest estimated system-speedups when running simultaneously with *p*. Periodically a user-space tool reads this data for all processes and greedily selects pairs with the highest estimated system-speedup. The tool feeds this plan back to the scheduler which heavily biases goodness in order to approximate gang-scheduling of the planned pairs. Any processes not in the plan, or those created after the planning cycle, will still run when processes in the plan block or exhaust their time-slice. For both algorithms the process time-slices are respected so starvation avoidance and static priorities are still available.

4 Evaluation

The evaluation machine was an Intel SE7501 based 2.4GHz Pentium 4 Xeon system with 1GB of DDR memory. RedHat 7.3 was used, the benchmarks were compiled with gcc 2.96. A single physical processor with Hyper-Threading enabled was used for the experiments. The scheduling algorithms were evaluated with sets of benchmark applications from the SPEC CPU2000 suite:

Set A: 164.zip, 186.crafty, 171.swim, 179.art.

Set B: 164.zip, 181.mcf, 252.eon, 179.art, 177.mesa.

Set C: 164.zip, 186.crafty, 197.parser, 255.vortex, 300.twolf, 172.mgrid, 173.applu, 179.art, 183.quake, 200.sixtrack.

Each benchmark application was run in an infinite loop. Each experiment was run for a length of time sufficient to allow each application to run at least once. The experiment was repeated three times for each benchmark set. The individual application execution times were compared to exclusive mode times to get a performance ratio similar to that described above. The sum of the performance ratios for the benchmarks in the set gives a

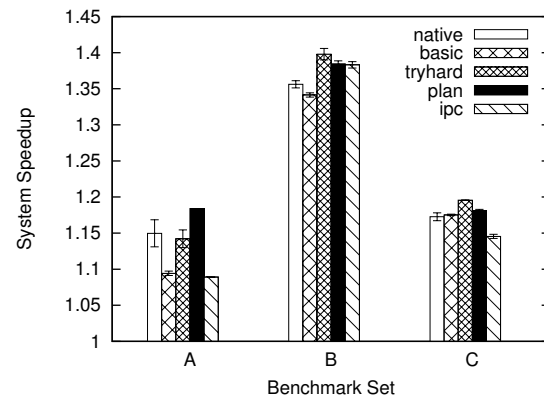


Figure 1: System-speedups for benchmark sets running under the different scheduling algorithms.

system-speedup. The execution time for the entire application, rather than a finer granularity, was used in order to assess the over all effect of the scheduling algorithm adapting to the effects of Hyper-Threading.

Each benchmark set was run with both *tryhard* and *plan*; the stock Linux 2.4.19 scheduler “*native*”; the same modified to provide physical, rather than logical, processor affinity “*basic*”; and an IPC-maximizing scheme using rolling-average IPC for each process and a goodness modification to greedily select tasks with the highest IPC (inspired by Parekh *et al*’s “G-IPC” algorithm [7]).

Figure 1 shows the system-speedups (relative to running the tasks sequentially) for each benchmark set with each scheduler. Improvements over *native* of up to 3.2% are seen; this figure is comparable with other work in the field and is a reasonable fraction of the 20% mean speedup provided by Hyper-Threading itself. The *tryhard* scheduler does reasonably well on benchmark sets B and C but results in a small slowdown on A: the four applications execute in a lock-step, round-robin fashion which *tryhard* is unable to break. It results in the same schedule as *native* but suffers the overhead of estimating performance. This is an example of worst-case behaviour that would probably be mitigated with a real, changing workload. *plan* provides a speedup on all sets.

The fairness of the schedulers was tested by considering the variance in the individual performance ratios of the benchmarks within a set. *tryhard*, *plan* and *basic* were as fair as, or fairer than *native*. The per-process time-slices were retained which meant that applications with low estimated performances were able to run once the better pairings had exhausted their time-slices. As would be expected, *ipc* was biased towards high-IPC tasks, however, the use of process time-slices meant that complete starvation was avoided. The algorithms were also tested for their respect of static priorities (“nice” values); both *plan* and *tryhard* behaved correctly. This

behaviour is a result of retaining the time-slices; a higher priority process is given a larger time-slice. Again, *ipc* penalized low-IPC processes but this was partially corrected by the retention of the time-slice mechanism.

5 Related Work

Parekh *et al* introduced the idea of thread-sensitive scheduling [7]. They evaluated scheduling algorithms based on maximizing a particular metric, such as IPC or cache miss rates, for the set of jobs chosen in each quantum. The algorithm greedily selected jobs with the highest metric; there was no mechanism to prevent starvation. They found that maximizing the IPC was the best performing algorithm over all their tests. Snavelly *et al*'s "SOS" (sample, optimize, symbiosis) "symbiotic" scheduler sampled different combinations of jobs and recorded a selection of performance metrics for each *jobmix* [8]. The scheduler then optimized the schedule based on this data executed the selected jobmixes during the "symbiosis" phase. Nakajima and Pallipadi used a user-space tool that read data from processor performance counters and changed the package affinity of processes in a two package, each of two Hyper-Threads, system [6]. They aimed to balance load, mainly in terms of floating point and level 2 cache requirements, between the two packages. They measured speedups over a standard scheduler of approximately 3% and 6% on two test sets chosen to exhibit uneven demands for resources. They only investigated workloads with four active processes, the same number as the system had logical processors. The technique could extend to scenarios with more processes than processors however the infrequent performance counter sampling can hide a particularly high or low load of one of the processes sharing time on a logical processor.

6 Conclusion and Further Work

We have introduced a practical technique for introducing awareness of the performance effects of Hyper-Threading into a production process scheduler. We have demonstrated that throughput gains are possible and of a similar magnitude to alternative user-space based schemes. Our algorithms respect static priorities and starvation avoidance. The work on these algorithms is ongoing. We are investigating better performance estimation methods and looking at the sensitivity of the algorithms to the accuracy of the estimation. We are considering implementation modifications to allow learned data to be inherited by child processes or through subsequent instantiations of the same application.

The scheduling heuristics were demonstrated using the standard Linux 2.4 scheduler – a single-queue dynamic priority based scheduler where priority is calculated for each runnable task at each rescheduling point. Linux

2.6 introduced the "O(1)" scheduler which maintains a run queue per processor and does not perform goodness calculations for each process at each reschedule point. The independence of scheduling between the processors complicates coordination of pairs of tasks. We plan to investigate further how our heuristics could be applied to Linux 2.6.

Acknowledgements

We would like to thank our shepherd, Vivek Pai, and the anonymous reviewers. James Bulpin was funded by a CASE award from EPSRC and Marconi Corp. plc.

References

- [1] J. R. Bulpin and I. A. Pratt. Multiprogramming performance of the Pentium 4 with Hyper-Threading. In *Third Annual Workshop on Duplicating, Deconstruction and Debunking (at ISCA'04)*, pages 53–62, June 2004.
- [2] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 5(1):1–13, Feb. 2001.
- [3] Intel Corporation. *Introduction to Hyper-Threading Technology*, 2001.
- [4] D. Koufaty and D. T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–64, 2003.
- [5] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(2):1–12, Feb. 2002.
- [6] J. Nakajima and V. Pallipadi. Enhancements for Hyper-Threading technology in the operating system — seeking the optimal scheduling. In *Proceedings of the 2nd Workshop on Industrial Experiences with Systems Software*. The USENIX Association, Dec. 2002.
- [7] S. S. Parekh, S. J. Eggers, H. M. Levy, and J. L. Lo. Thread-sensitive scheduling for SMT processors. Technical Report 2000-04-02, University of Washington, June 2000.
- [8] A. Snavelly and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, pages 234–244. ACM Press, Nov. 2000.
- [9] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '2003)*, pages 26–34. IEEE Computer Society, Sept. 2003.
- [10] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th International Symposium on Computer Architecture (ISCA '95)*, pages 392–403. IEEE Computer Society, June 1995.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

SAGE

SAGE is a Special Interest Group (SIG) of the USENIX Association. It is organized to advance the status of computer system administration as a profession, establish standards of professional excellence and recognize those who attain them, develop guidelines for improving the technical and managerial capabilities of members of the profession, and promote activities that advance the state of the art or the community.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

- ❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖ AMD ❖ Asian Development Bank ❖
- ❖ Atos Origin BV ❖ Cambridge Computer Services, Inc. ❖ Delmar Learning ❖
- ❖ DoCoMo Communications Laboratories USA, Inc. ❖ Electronic Frontier Foundation ❖
- ❖ Hewlett-Packard ❖ IBM ❖ Intel ❖ Interhack ❖ MacConnection ❖
- ❖ The Measurement Factory ❖ Microsoft Research ❖ Oracle ❖ OSDL ❖ Perfect Order ❖
- ❖ Portlock Software ❖ Raytheon ❖ Sun Microsystems, Inc. ❖ Taos ❖ Tellme Networks ❖
- ❖ UUNET Technologies, Inc. ❖ Veritas Software ❖

SAGE Supporting Members

- ❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖ Asian Development Bank ❖
- ❖ Fotosearch ❖ Microsoft Research ❖ MSB Associates ❖ Raytheon ❖
- ❖ Ripe NCC ❖ Taos ❖ Tellme Networks ❖

For more information about membership, conferences, or publications,
see <http://www.usenix.org/>
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: office@usenix.org

ISBN 1-931971-27-7